

Exploiting non-determinism for reliability of mobile agent systems

Ajay Mohindra, Apratim Purakayastha
IBM Thomas J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
{ajaym,apu}@us.ibm.com

Prasanna Thati
Department of Computer Science
University of Illinois
Urbana, IL 61801
thati@cs.uiuc.edu

Abstract

An important technical hurdle blocking the adoption of mobile agent technology is the lack of reliability. Designing a reliable mobile agent system is especially challenging since a mobile agent is potentially affected by failure of any host that it visits, or failure of any communication link that it needs to traverse. Previous work in this domain has attempted techniques such as periodic checkpointing of mobile agent state and restarting upon machine or communication recovery. Such approaches render an agent unavailable until a machine or a communication link itself recovers. In this paper, we take an alternate approach based on the premise that a mobile agent can often complete its task in more than one way. We capture such redundancy in non-deterministic constructs in the agent language and maintain state about an agent's actual computational path in its possible computational tree. We design and implement a distributed recovery scheme that detects a failure, rolls back an agent's computation, and restarts the agent from a previous point in its computational tree down a different but equivalent computational path without waiting for the actual failure itself to be repaired.

1 Introduction

Mobile agents are useful for certain tasks such as searching and filtering information, and monitoring remote events. They are also appropriate for networks with low bandwidth, high latency, and intermittent connectivity. Although numerous mobile agent systems have been proposed [2, 4, 7, 12, 14, 15], they have not been widely used in practice. One significant technical concern regarding mobile agent systems is their lack of reliability. In this paper, we focus on the reliability aspects of mobile agent systems. We formulate,

design, and implement a mobile agent system that exploits non-determinism in the agent code to achieve greater reliability.

A mobile agent potentially executes on a number of different hosts migrating from one to the other. Building a reliable mobile agent system is especially difficult since failure of any one of the hosts or failure of communication between any two hosts may adversely affect the mobile agent. A few systems (see Section 2), have proposed periodic checkpointing of mobile agent state and restoration of the agent upon recovery of the failed host. In such systems, the mobile agent cannot proceed until the failed host has recovered. Other systems have proposed using message queuing for reliable agent migration. In such systems, the agent cannot proceed if the queue manager fails or loses communication.

Our approach is based on the assertion that a mobile agent can often perform its task in more than one way. For example, a search agent can retrieve pertinent information by using one information source or another. In this model, a mobile agent makes *choices* as it proceeds with its computation. If a failure occurs after making a certain choice, our system detects the failure and restarts the mobile agent with a different choice. This approach allows the mobile agent to proceed with its computation by routing the agent around failures in contrast with previous approaches where the mobile agent has to wait for the failures themselves to be repaired. We design and implement the reliability mechanisms in the context of *NetPebbles* [9], a mobile agent system that allows scripting with network components. The agent language in *NetPebbles* has non-deterministic constructs which we exploit for the purposes of reliability.

The rest of the paper is organized as follows. Section 2 discusses related work and identifies the key differences in our approach from other approaches. Section 3 outlines the *NetPebbles* programming model, introduces its non-deterministic constructs, and illus-

trates how such non-determinism can be exploited for greater reliability. Section 4 describes our distributed recovery scheme consisting of failure detection, roll-back, and garbage collection mechanisms. Section 5 presents implementation and a preliminary evaluation of our system. Finally, Section 6 concludes and outlines future work.

2 Related Work

A number of mobile agent systems have explored reliability issues. The Tacoma [12] and the Ara [11] systems counter processor failures by providing checkpointing primitives whereby an agent can save its state. The Voyager [4] and Concordia [15] systems also provide checkpointing facilities to deal with processor failures. In these systems, a mobile agent cannot proceed with its computation until the failed host recovers and restores the state of the mobile agent from persistent storage. Moreover, these systems also require frequent checkpointing at all hosts that the mobile agent visits. In Concordia, reliable agent migration is realized using a message queuing system for communication. If the queue manager fails or loses communication, the agent is delayed until the queue manager recovers. In contrast with the above, our system can recover and restart a mobile agent from a previous execution context, independent of the recovery of a failed host or a communication link. In the context of computer-aided manufacturing, Wolfson *et al* propose “intelligent routers” that are similar to mobile agents [17]. They use the ISIS [1] distributed programming environment for fault tolerance of intelligent routers in a local area network. In [5], Johansen *et. al* propose a detection and recovery protocol for implementing fault-tolerant itinerant computations. Mobile agent systems can be deployed on ISIS-like distributed programming environments, but it is not clear how such systems scale beyond local area networks.

Strasser *et al* [13, 14] discuss two approaches for improving reliability in agent systems. The first approach allows an agent to specify a flexible *itinerary* with the possibility to defer the visit to currently unavailable machines or to select alternate machines in case of machine failures. The second approach uses a fault-tolerant protocol to implement *exactly-once* execution property for agents using formation of explicit *stages* of computation and results in a constrained lock-step manner of execution. Such execution constraints, although useful for certain classes of applications, are not desirable in general. In [16], Wang *et al* describe a method of bypassing software faults by exploiting available non-determinism in message deliveries at runtime.

In contrast, our approach exploits component and task level redundancies to bypass faults.

Using non-determinism to improve reliability of mobile agents is certainly inspired by well-known fault tolerance techniques that use temporal redundancy of software modules [10]. The domain of mobile agents, however, introduces some unique challenges which must be addressed. First, the migratory nature of mobile agents makes it difficult to ascertain faults and initiate recovery. Second, the process of recovery not only includes starting the agent from a checkpoint at some previous host, but may also include garbage collecting the residual state of the mobile agent that persists at other hosts visited by the agent after the checkpoint operation.

```

dim fundYield[5]
fund = "GROWTH"
a = createComponent("IMutualFund", "name = "+fund)
fundYield = a.getAvgAnnualReturn(fund, 5)
c = createComponent("IDisplay") at "mypc.com"
c.plot(fund, fundYield)
exit

```

Figure 1. An example script that locates a Mutual Fund data component to determine the average annual rate of return for the last five years for a specific fund. NetPebbles’s constructs are shown in **boldface**.

3 NetPebbles Programming Model

The NetPebbles environment offers a component-based programming model. A NetPebbles programmer writes a script by first selecting required interfaces from a catalog and then invoking interface methods as if the components implementing the interfaces are local. An end user simply starts a script and thereafter the NetPebbles runtime dynamically determines the component sites and transparently moves the state of the script to the component sites as necessary. When the script execution completes, the script returns to the starting site with the results. For a comprehensive discussion of the NetPebbles programming model, please refer to [9].

3.1 An Example script

We illustrate the NetPebbles programming model using an example script. Figure 1 shows a script that determines the average annual return for a mutual fund

for the last five years. At the time of writing the script, the programmer knows that the script needs to use two interfaces, namely, “IMutualFund”, and “IDisplay”. The “IMutualFund” interface implements a method that provides the average annual return for a mutual fund. Each component that implements this interface may only provide data for a selected set of funds. The “IDisplay” interface provides methods to plot the results. For brevity, we ignore error conditions in the script.

In the first `createComponent()` call, the runtime attempts to create a component that implements the “IMutualFund” interface and supports the specified fund. The runtime uses the component catalog to determine the location of the appropriate component and migrates the script to that location. The runtime at the new location instantiates the component and invokes the “getAvgAnnualReturn()” method on the component instance with the fund name and number of years as arguments. The results from the method invocation are stored in the array variable “fundYield”. The script then uses the component catalog to determine the location of the component that implements the “IDisplay” interface. When a component is found, the runtime migrates the script to the location specified by the `at` construct (“mypc.com”) and downloads the component to that location. The runtime at the new location creates an instance of the component and displays the results by invoking the “plot()” method. Finally, with the `exit` statement, the script completes execution, and the runtime initiates garbage collection of all component instances.

3.2 Non-determinism in NetPebbles

The NetPebbles programming model is based on the premise that there is more than one way to arrive at the correct result. To explore this premise, we added two programming constructs that exploit non-determinism. The two constructs are “`createComponent`” (as described in Section 3.1) and “`any`”. We briefly describe each of these constructs below. The list is not exhaustive, and based on our experience with these two constructs, we plan to add more constructs, such as `asmanyas` and `atleast(N)`, in the future. Note that the focus of our work is *not* to explore novel language constructs but the underlying fault-tolerance mechanisms (see Section 4) to support such constructs.

createComponent construct: In NetPebbles, more than one component may be available that implements the same interface, i.e., provides semantically equivalent function. When the NetPebbles runtime, say at L_1 , resolves an interface name from the com-

ponent catalog, the catalog may return more than one location where the desired component is available. The runtime makes a non-deterministic *choice*, selects one location, say L_2 , from the list and tries to migrate the script to L_2 . The runtime at L_1 may be unable to migrate the script due to the failure of host L_2 or failure of the communication link between L_1 and L_2 . Even after successful migration to L_2 , a failure may still occur during component instantiation or method execution. When such a failure is detected by L_1 , the runtime makes another selection from the list of component locations and restarts the script. Figure 1 shows an example script that uses the `createComponent` construct.

Any construct: While the `createComponent` construct provides component-level redundancy in NetPebbles, the `any` construct provides task-level redundancy. The `any` construct is similar to the ALT construct in Occam [6]. It consists of a set of *tasks* enclosed by boolean *guards*. The `any` construct allows the script to proceed with the computation as soon as any one of the guards evaluates to “true”. If more than one guard evaluate “true” then the runtime makes a non-deterministic *choice* and executes the statements contained in any one of the guard blocks. If during the execution of these statements, the script encounters a failure, the NetPebbles runtime rolls back the script and chooses another task whose guard evaluates to “true”. If there is no such task, then the `any` construct fails. Figure 2 shows an example that uses the `any` construct. In the example, the script computes the average annual return for a mutual fund for last five years, and compares the result with the returns for the market, using either the Russell2000 index or the S&P500 index. In the example, both the guards evaluate to “true” and the runtime makes a *choice* between the two possibilities. After obtaining the returns for an index, the script displays the comparison at machine “mypc.com”.

In this paper, we discuss the use of non-determinism in the context of NetPebbles. The non-deterministic constructs that we use, however, are generic in nature and can easily be applied to other mobile agent systems. The `createComponent` construct essentially enables an agent to choose a host among several hosts that offer the same function. In other systems such as Aglets [7] or Agent-Tcl [2], an explicit migration step such as a `dispatch` method or an `agent_jump` command is used to migrate an agent from one site to another. In these systems, it is straightforward to augment these instructions with a multiple destination site names such that the agent can choose to migrate to any of these sites to accomplish its goal. Our approach can then be applied by non-deterministically

```

dim fundYield[5]
fund = "GROWTH"
a = createComponent("IMutualFund", "name="+fund)
fundYield = a.getAvgAnnualReturn(fund, 5)
dim indexYield[5]
any
  case (true)
    index = "Russell2000"
    b = createComponent("IMarketIndex", "name="+index)
    indexYield = b.getAvgAnnualReturn(index,5)
  endcase
  case (true)
    index = "S&P500"
    b = createComponent("IMarketIndex", "name="+index)
    indexYield = b.getAvgAnnualReturn(index,5)
  endcase
endany
c = createComponent("IDisplay") at "mypc.com"
c.plot(fund, fundYield, index, indexYield)
exit

```

Figure 2. A script that uses the **any** construct to compare the average annual return of a mutual fund to that of Russell 2000 index or the S&P500 index. NetPebbles's constructs are shown in **boldface**.

choosing from the list of specified hosts, and recovering and retrying another host upon detecting a failure. The task level non-deterministic constructs which we explore are equally applicable to any other mobile agent system. For example, a group of Agent-Tcl commands can be placed under the **any** construct if such a construct were to be added to the Agent-Tcl vocabulary. Our system can then non-deterministically choose to evaluate any task block in the same manner as in NetPebbles.

4 The Distributed Recovery Scheme

In the NetPebbles system a script migrates from one machine to another during its execution. During its lifetime, a script can encounter a variety of failures such as node failures, communication-link failures, and component failures. The effect of any such failure is the same – the script cannot continue its execution. To handle failures, the NetPebbles runtime uses non-determinism to route around failures instead of waiting for failures themselves to heal.

While executing a non-deterministic construct in a script, the runtime makes a *random choice* from a set of viable alternatives. The choice can either be a component-level choice as made by the **createComponent** construct or a task-level choice as made by the **any** construct. We refer to the script state when a choice is made as a *choice point*, and the cur-

rent host as the *choice host* for that choice point. Each possible choice at a choice point is assigned a unique identifier known as the *choice id*. The ordered collection of all past *choice points* in a script's lifetime is its *choice history*. A script's choice history is abstractly represented as a vector of choice ids that identify the choices the script has made so far. Similarly, a choice point is abstractly represented as a vector of choice ids that identify the choices until and including the one at the choice point. A script carries its choice history and the vector of corresponding choice hosts along with it.

The recovery strategy of NetPebbles is to find a fault-free path in a script's computation tree. Figure 3 illustrates this concept. Points marked by *A*, *A'*, and *A''* correspond to *choice points*, and each outgoing edge at these nodes represents a choice. When a failure, denoted by a cross (X in Figure 3a), is detected by the runtime, the script is rolled back to the most recent *choice point A'*, and an alternate choice is made (Figure 3b). If the new choice also leads to a failure, the current choice at *A* is declared faulty and the script is rolled back and restarted with a different choice at *A* (Figure 3c).

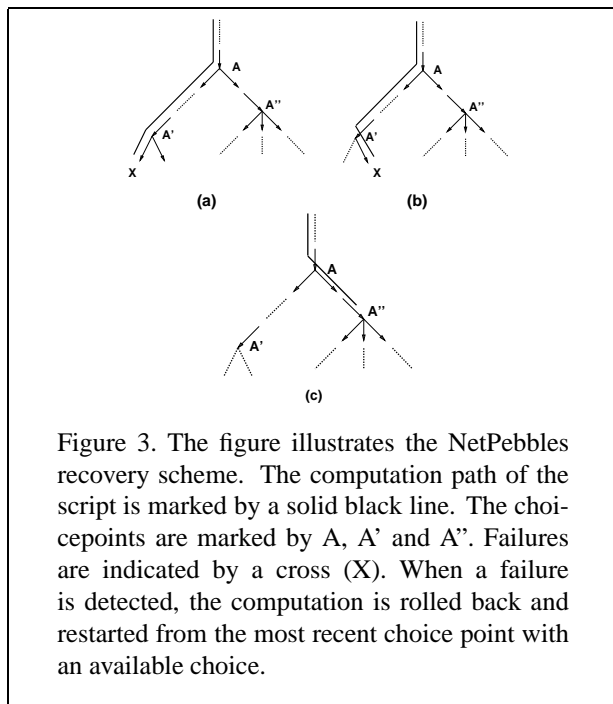


Figure 3. The figure illustrates the NetPebbles recovery scheme. The computation path of the script is marked by a solid black line. The choice points are marked by *A*, *A'* and *A''*. Failures are indicated by a cross (X). When a failure is detected, the computation is rolled back and restarted from the most recent choice point with an available choice.

The distributed recovery scheme uses three underlying services at each node in the system: failure detection, rollback, and garbage collection. The failure detection service monitors scripts that have made a choice at the local node and detects failed choices. The rollback service saves a script's state at the local node

before every choice made by the script, and when notified of a failed choice, it restores the appropriate state and restarts the script with a different choice. The garbage collection service actively tracks a migrating script and initiates garbage collection at all appropriate hosts when informed of a failed choice.

4.1 Failure Detection

In our failure detection scheme, a script is monitored at every choice point in its choice history by the corresponding choice hosts to detect any failed choices. A script's choice at a choice point is declared failed if all the choice points it leads to and the script have failed. A choice point fails if the corresponding choice host fails, or all choices at the choice point have failed. The script fails if its current host fails, or it is unable to migrate because of a network partition, or one of its components at the current host has failed.

Given the fact that exact failure detection in an asynchronous system with arbitrary host and communication failures is impossible [8], our aim is to approximate this scheme as accurately and efficiently as possible. In particular, our specific design goals are to recover from multiple faults and network partitions, minimize message complexity, minimize time to initiate recovery, and minimize false failure detection (declaring a fault when there is none).

There are several approaches for failure detection. In the simplest approach, every choice point (the corresponding choice host) and the script's current host send periodic heartbeats to all predecessor choice points in the script's choice history. A choice point declares its current choice as failed if it does not receive any heartbeat during a pre-configured time interval. This naive approach has a prohibitive $O(n^2)$ message complexity, where n is the number of choice points in a script's choice history.

Our current approach is a simple modification of the above scheme. A choice point and the script send regular heartbeats only to its immediately preceding choice point in the script's choice history. For choice points that are further up in the choice history, heartbeats are sent with decreasing probability. More precisely, assuming n choice points CP_1, \dots, CP_n in a script's choice history, CP_i sends periodic heartbeats to CP_j for $1 \leq j < i$ with probability r^{i-j-1} , where $0 \leq r \leq 1$. The script sends heartbeats to each CP_i for $1 \leq i \leq n$ with probability r^{n-i} . As before, a choice host initiates recovery if and only if it does not receive any heartbeat in a pre-specified period. Figure 4 illustrates this scheme.

This probabilistic scheme has a tunable message

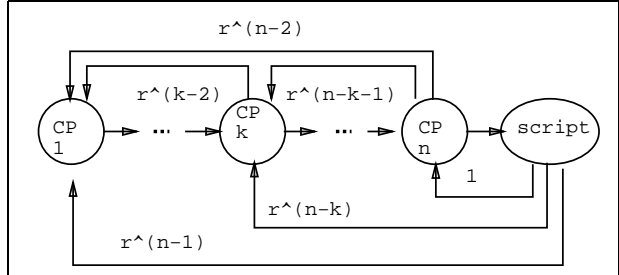


Figure 4. This figure illustrates the flow of heartbeats between a script and the choice points in the script's choice history. The script and its choice points send heartbeats to each previous choice point in the script's choice history with geometrically decreasing probability. Here r is the probability of sending a heartbeat and n is the number of choice points in a script's choice history.

complexity and accuracy. In fact, the naive approach is a special case of this scheme with $r = 1$. For false failure detection to occur, a choice point has to miss heartbeats from all the successor choice points and the script's current host. Therefore, the probability of such a false failure detection at CP_i is

$$\prod_{i < j \leq n+1} [(1 - r^{(j-i-1)}) + pr^{(j-i-1)}]$$

assuming p is the probability of a message loss. Since $p \leq 1$, smaller the r , higher the probability of false failure detection. For $r < 1$, the expected number of messages sent per period is $\sum_{i=1}^{n+1} \sum_{j=1}^{i-1} r^{i-j-1}$ which evaluates to

$$\frac{n + 1 - \sum_{i=1}^{n+1} r^{i-1}}{1 - r}$$

Thus message complexity varies from $O(n)$, when $r = 0$, to $O(n^2)$, when $r = 1$.

4.2 Checkpointing and Rollback

The rollback service at a node maintains in a table an association between the choice histories of the script that it has observed in the past and the script's local state. When notified of a script's failed choice, the rollback service uses the information to reset the script's local state to what it was just before the choice was made.

The table is initially empty and is modified on two occasions. First, when a local script executes a non-deterministic construct, a new choice id is generated for

the new choice and is appended to the script's choice history. Information associating the new choice history and the execution state of the script and its local components is added to the table. Second, when a node receives a script with a previously unobserved choice history, the state of the script's local components is checkpointed. The state is then associated with the received choice history and this information is added to the table.

When notified of a failed choice, the rollback service searches the table for the choice history that has the failed choice point as a prefix and is minimal according to the prefix relation, and restores the state associated with the choice history. The rollback service also notifies the local components to undo any changes that need special handling. Further, if the failed choice was made locally then restoring the local state automatically resumes execution of the script from the non-deterministic statement that made the choice. The stored information about the failed choice is used to make a different choice.

The use of a rollback scheme for recovery has implications on the types of components that a script can use. The state of each component needs to be *rollbackable*, that is, either the component supports methods that are idempotent or the component supports additional methods that would allow the effects of non-idempotent methods to be undone. On surface, such requirements may seem to be too constraining for component developers, but in reality many components already exhibit such properties. For example in the system administration domain, components that read system state are idempotent in nature, and components that update system state also support actions that undo the changes. Examples of the latter include components that *install/un-install* software, *start/kill* daemons, and *enable/disable* file permissions.

4.3 Garbage Collection

When a script completes execution, it returns to its starting host. The garbage collection service at the starting host initiates a clean up of the script's distributed state by informing all the nodes visited by the script. A script carries along with it a list of nodes that it has visited. This list is updated as the script migrates in the network.

The garbage collection service also helps rollback the distributed state of a script when one of its choices fail. When notified of a failed choice by the local failure detection service, the garbage collection service notifies the rollback service at all the nodes that the script visited after the choice. This notification effectively

resets the script's state at all nodes to what it was before the choice. Identifying the nodes which need to be notified requires explicit tracking of the script. Note that when a choice fails, the script need not be present at the corresponding choice host. In order to track scripts, when a script migrates to a remote node, the local garbage collection service saves the script's choice history and remote node address in a persistent store.

Network partitions may delay garbage collection indefinitely. To alleviate this problem, the garbage collection service may use a policy of automatically garbage collecting the local script state after a certain period of inactivity.

4.4 False Failure Detection and Multiple Script Incarnations

The failure detection service may declare a choice to have failed even if it hasn't. Such false detections are possible because of arbitrary communication delays and losses. As a result, multiple incarnations of the same script may execute simultaneously in the system. Multiple incarnations may not always affect program correctness, although they waste system resources. A script whose execution has no side effects, the results returned by all but one of the incarnations can simply be ignored. A script that only performs idempotent operations over the network is an example of a script with no side effects. On the other hand, multiple incarnations of scripts that cause side effects can affect program correctness if they are not handled appropriately. A system administration script that modifies the machine state is an example of a script with side effects.

Multiple script incarnations are handled in the NetPebbles system as follows. Since the garbage collection service actively tracks a script, the cleanup messages sent by it after a false rollback eventually catch up with the obsolete incarnation. When a rollback service is informed of a failed choice of a script, and it finds a local incarnation of the script with the failed choice id in its choice history, the garbage collection service kills the obsolete incarnation.

The cleanup messages sent by the garbage collection service after a false rollback may take considerable time to catch up with obsolete incarnations. Meanwhile, obsolete incarnations should not interfere with the execution of the latest incarnation. The NetPebbles system ensures this non-interference as follows. All script related messages such as migration or garbage collection messages contain the script's choice history. When a node receives a message, it checks if the received choice history is obsolete by comparing the choice history with

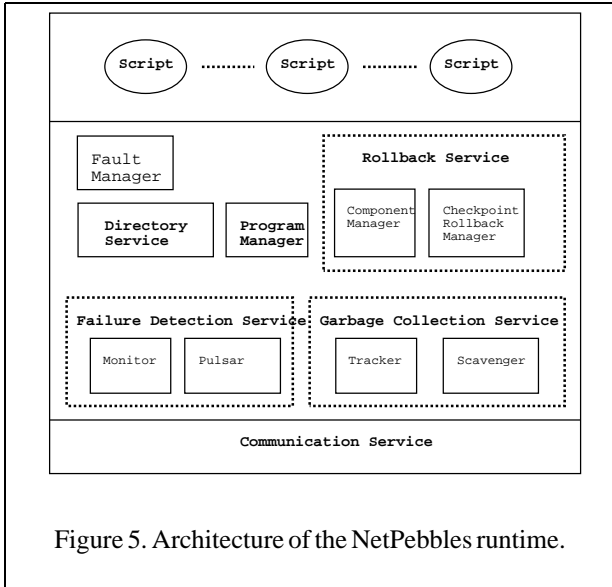


Figure 5. Architecture of the NetPebbles runtime.

the script’s latest choice history known to the node. When a node receives a message with an obsolete choice history, it simply discards the message. In particular, when a node receives an obsolete incarnation (a migration message), it does not entertain the incarnation. This effectively kills the obsolete incarnation.

5 Implementation and Preliminary Evaluation

In this section, we first describe the implementation of NetPebbles, with emphasis on its distributed recovery scheme; we then present preliminary experimental results and discuss the performance and scalability of the implementation.

5.1 Implementation Overview

Figure 5 shows the architecture of the NetPebbles runtime. The NetPebbles runtime has been implemented in Java. It consists of a Program Manager, a Fault Manager, a Directory Service, a Rollback Service, a Failure Detection Service, a Garbage Collection Service, and a Communication Service.

When a NetPebbles runtime is first started, it initiates a scheduler that performs pre-emptive round-robin scheduling of all threads in the system. The scheduler is necessary to ensure fair thread scheduling in the heavily threaded NetPebbles runtime¹. The system then starts the threads related to the program manager, failure detection, garbage collection and rollback

¹Thread scheduling in some JVMs can be unfair and may cause starvation.

services. We now describe each of the components and the interactions between them.

Program Manager: This module includes an interpreter for executing scripts, and maintains the execution context of scripts that have visited the local node. It also generates globally unique identifiers for scripts launched at the local node. When a program manager receives a script, it modifies the script’s list of visited nodes, associates an existing execution context with the script (if the script is revisiting the node) or creates a new one (for a new script), and inserts a program thread into the scheduler’s queue. The program thread is an interpreter which repetitively extracts the next element from a script’s statement stack and executes it. The statement stack essentially plays the role of program counter and is modified appropriately by the execution of the topmost statement. The interpreter informs the fault manager about executing a non-deterministic statement. The fault manager then extends the script’s choice history and checkpoints the script along with the script’s stack, data, and local components. Execution of a statement may also result in script migrations; for example, when a `createComponent` call creates a component at a remote node. During migration, the script’s stack and data is serialized along with other system level information such as the script’s choice history and list of visited nodes, and sent to the intended host. When a script completes execution, it is sent back to its starting host. The program manager also informs the fault manager about arrivals and departures of all scripts.

Directory Service: This service provides an interface to a component catalog for resolving component creation requests from scripts. The component catalog has been implemented using the LDAP [3] distributed directory services.

Fault Manager: This module coordinates all fault-tolerance related activities. When a script arrives at a node, the fault manager checks to see if the script incarnation is obsolete (see Section 4.4). If the script incarnation is obsolete, the fault manager informs the program manager to discard the incarnation.

With the help of the rollback service, the fault manager maintains for each script, an association between the script’s observed choice histories and the script’s local state (see Section 4.2). When a script arrives at the local node, the fault manager informs the failure detection service to send heartbeats (see Section 4.1) to all choice points in the script’s choice history. These heartbeats are cancelled when the script departs. Further, when a script executes a non-deterministic construct, the fault manager informs the failure detection

service to monitor the new choice. When informed of a failed choice, the fault manager uses the garbage collection service to restore the script's distributed state to what it was just before the choice was made.

Failure Detection Service: This service consists of a pulsar thread and a monitor thread that continuously process requests in their respective queues. A request in the pulsar's queue specifies a heartbeat targeted at a particular choice point of a script. The request specifies the target, probability and scheduled time for a heartbeat. The target is a triplet of choice host address, script id, and choice point. A request in monitor's queue specifies the time by which the next heartbeat for a particular choice point is to be received. The monitor sleeps until it is interrupted by a broken deadline or the receipt of a heartbeat. If the monitor misses a heartbeat, it signals a failure to the fault manager. Since heartbeats are periodic, both the pulsar and monitor reschedule a request after sending or receiving a heartbeat.

Rollback Service: This service consists of a component instance manager, and a checkpoint and a rollback manager. The former maintains associations between scripts and components created by the scripts at the local host. It also generates global names that are used to refer to components uniformly across different nodes. The checkpoint and rollback manager is used by the fault manager to store and retrieve a script's local state (see Section 4.2).

Garbage Collection Service: This service consists of two threads: a tracker and a scavenger. The tracker maintains information in a persistent store to track scripts that migrated away from the local node. The scavenger processes "cleanup" requests received from other hosts. If the request is in response to the completion of a script then the scavenger cleans up all script related information at the local node. If the request is in response to a failed choice, the scavenger informs the fault manager to rollback the script's local state to the appropriate choice point. The garbage collection service propagates the cleanup message to remote nodes using information maintained by the tracker.

5.2 Preliminary Experimental Results

In this section, we present the performance results of our distributed recovery scheme. We perform two sets of experiments. The first set measures "null overhead" of the system, where we estimate the overhead of simply running the fault tolerance machinery even if there are no actual failures in the system. The second set measures the recovery time of our system when

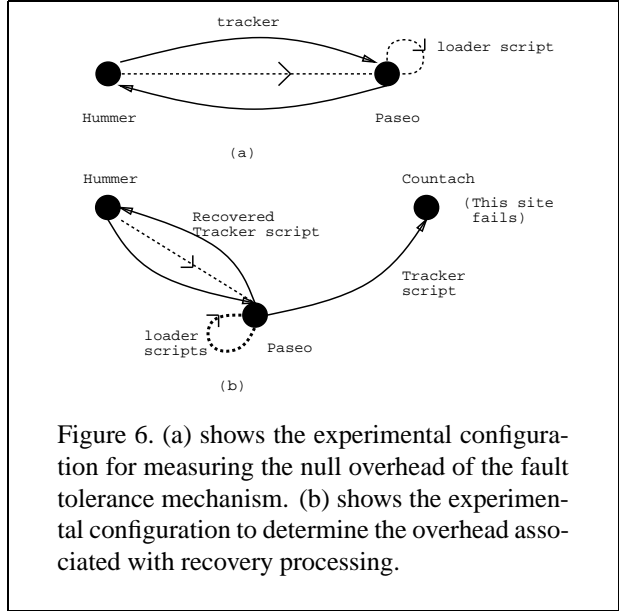


Figure 6. (a) shows the experimental configuration for measuring the null overhead of the fault tolerance mechanism. (b) shows the experimental configuration to determine the overhead associated with recovery processing.

failures actually happen.

Experimental configuration: We use three machines for the different experiments conducted: an IBM ThinkPad running Windows 98 ("hummer"), and two IBM PCs running Windows NT ("paseo" and "countach"). The machines are Pentium II class machines connected via a 16MB/sec token ring network. Figure 6a shows the experimental configuration for the first set of experiments. We use a number of "loader" scripts that start from the machine "hummer", migrate to the machine "paseo", executing an infinite *while loop* with alternate sleep and wakeup times. Increasing the number of such loader scripts simulates increasing *load* on the system. When fault tolerance is enabled, loading the system in such a fashion results in monitors being scheduled on "hummer", and heartbeats being scheduled on "paseo" that are to be sent to "hummer". When a system is loaded by a certain number of loader scripts, a "tracker" script is sent from "hummer" to "paseo", and returned immediately back to "hummer", where we measure the *roundtrip time* for the tracker script. We measure the roundtrip time under various load conditions with and without enabling fault tolerance and thereby measure the overhead of fault tolerance under various load conditions. Note that the measurements report the elapsed "wall-clock" time.

Figure 6b shows the experimental configuration for the second set of experiments. We used the loader scripts exactly in the same manner as in the first set of experiments. However, in this case, the "tracker" script is first sent from "paseo" to "countach". When the tracker script reaches "countach", a failure is sim-

ulated on “countach” by aborting the runtime. The monitor for the tracker script on “paseo” eventually detects the failure and starts the recovery process for the script. Upon recovery, another choice in the tracker script gets activated and the script migrates to “hammer” and returns immediately to “paseo” to complete execution. In this experiment, we measure the *recovery time* on “paseo”. The recovery time is the time between the detection of the failure by the monitor thread and scheduling of the rolled back script for execution.

We use heartbeat probability of 1.0 for both sets of experiments to fully expose the overheads related to fault tolerance. The experimental load was varied from 0 to 100 loader scripts in steps of 20. Although we ran tests with up to 200 loader scripts, the NetPebbles runtime occasionally ran out of memory for loads higher than 100. We are confident that for load values of up to 100, the system is not affected by extraneous factors. We conducted 9 trials for each data point on the result graphs. For each data point, we discarded the two highest and the two lowest values to eliminate outliers, and computed the arithmetic average of the remaining 5 values. The coefficient of variations for measured data points was less than 0.05 in most cases,² giving us relatively high confidence in the computed averages.

Results: Figure 7 shows the overhead of our fault tolerance machinery for different system loads. For low system load (up to 20 loader scripts), the overhead of fault tolerance is less than 30%. For moderate loads (up to 40 loader scripts), the overhead of fault tolerance is roughly 60%. For heavy loads (up to 60 and higher loader scripts), the overhead of fault tolerance is roughly 100%. For heavy load cases, the processors are at their maximum utilization and there is heavy contention between concurrent threads (program threads, pulsar thread, monitor thread). The initial results from our implementation are not discouraging as we have increased the load by a factor of 5 (from 20 to 100) and the overhead has only increased up to a factor of 2.

Figure 8 shows the recovery time for a script with increasing system load. The recovery time is relatively low (a few tens of milliseconds) when the system is loaded lightly to moderately (up to 40 loader scripts). However, with loads of 60 or higher, the processors again reach maximum utilization and concur-

²In cases with low load since there was no competition for the processor, the execution times varied only a little. In cases with high load, since the processor was fully utilized, tracker threads always had to wait, and hence execution times varied only a little. In cases of moderate load, however, the execution time varied more as sometimes the tracker threads “got lucky” and were scheduled quickly while most loader threads were sleeping, and at other times they had to wait appreciably.

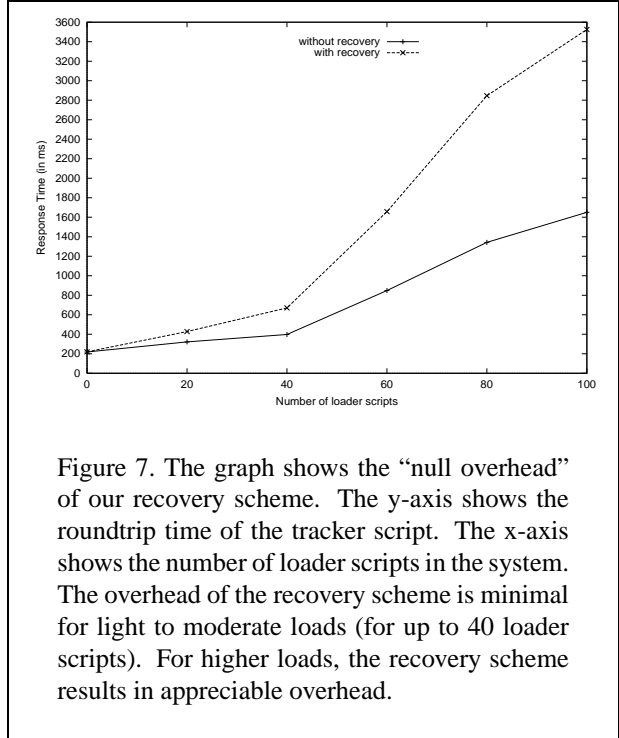


Figure 7. The graph shows the “null overhead” of our recovery scheme. The y-axis shows the roundtrip time of the tracker script. The x-axis shows the number of loader scripts in the system. The overhead of the recovery scheme is minimal for light to moderate loads (for up to 40 loader scripts). For higher loads, the recovery scheme results in appreciable overhead.

rent threads face tremendous contention. The recovery time increases roughly by a factor of 20 when the load increases from 40 to 100.

The initial performance results indicate that high system load must be managed carefully in our system. To expose all the overheads in our system, we fixed the heartbeat probabilities in our system at 1.0. Decreasing the heartbeat probability, and increasing monitoring timeout will likely make the system scale better. Techniques such as grouping heartbeats together will also likely improve system performance. To keep the initial implementation clean, we have also used Java language threads quite liberally in our system. Streamlining the usage of threads in our system, and possibly using native thread packages will likely improve system performance appreciably.

6 Conclusions and Future Work

In this paper, we describe a scheme that exploits non-determinism to address the issues of reliability and fault tolerance for mobile agents. First, we describe the non-deterministic constructs of the NetPebbles programming model and illustrate how mobile agents can use these constructs. Next, we describe a distributed recovery scheme that exploits non-determinism to provide fault tolerance. We present the design and implementation of the rollback scheme in context of the

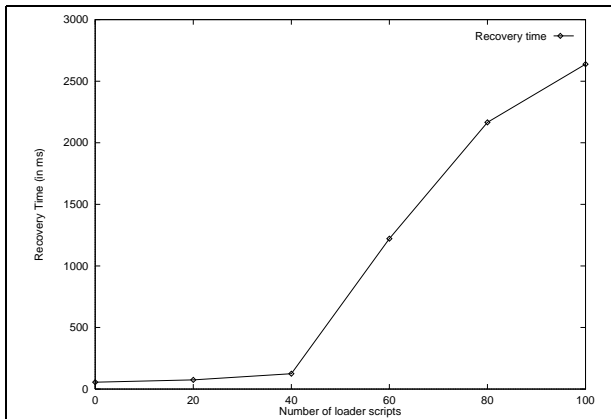


Figure 8. The graph shows the recovery time of the tracker script when a failure occurs. The y-axis shows the recovery time on the choice point host. The x-axis shows the number of loader scripts on the choice point host. The recovery time is minimal for light to moderate loads (for up to 40 loader scripts). For higher loads, the recovery time increases appreciably.

NetPebbles system along with some preliminary performance measurements. Our initial performance results are promising but indicate that high system load must be managed carefully.

In the future, we plan to investigate additional constructs for providing non-determinism in the programming model. We also plan to implement more sophisticated failure detection schemes such as ones that use voting. We plan to fine tune the implementation by making judicious use of Java language threads and possibly using native thread packages. We also plan to study the effects of decreasing the pulse probability, and increasing monitoring timeout on the overall system performance and scalability. Techniques such as grouping pulses together will also likely improve system performance.

Acknowledgments

We would like to thank Murthy Devarakonda and Yuanyuan Zhou for their valuable input on reliability and fault tolerance aspects of NetPebbles.

References

[1] K. P. Birman and T. A. Joseph. Reliable Communications in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[2] R. S. Gray. Agent Tcl: A Transportable Agent System. *Workshop of Intelligent Information Agents in Fourth International Conference on Information and Knowledge Management*, December 1998.

[3] T. Howes and M. Smith. A Scalable Deployable Directory Service for the Internet. In *Proceedings of INET 95*, 1995.

[4] O. Inc. Voyager. <http://www.objectspace.com/products/voyager>, 1998.

[5] D. Johansen, K. Marzullo, F. B. Schneider, K. Jacobsen, and D. Zagorodnov. NAP: Practical Fault-Tolerance for Itinerant Computations. Technical report, Cornell University, 1998.

[6] G. Jones. *Programming in Occam*. Prentice Hall International, 1987.

[7] D. B. Lange, M. Oshima, and O. Mitsuru. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley Publishing Co, 1998.

[8] N. Lynch. *Distributed Algorithms*. Morgan Kaufman Publishers, 1996.

[9] A. Mohindra, A. Purakayastha, D. Zukowski, and M. Devarakonda. Programming Network Components using NetPebbles: An Early Report. In *Fourth Annual Usenix Conference on Object Oriented Technologies and Systems*, April 1998.

[10] V. P. Nelson and B. Carroll. *Fault Tolerant Computing*. IEEE Computer Society Press, 1987.

[11] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *Proceedings of the First International Workshop on Mobile Agents*, 1997.

[12] F. B. Schneider. Towards Fault tolerant and Secure Agents. *11th International Workshop on Distributed Algorithms*, Germany, September, 1997.

[13] M. Strasser and K. Rothermel. Reliability Concepts for Mobile Agents. *International Journal of Cooperative Information Systems*, 7(4), 1998.

[14] M. Strasser, K. Rothermel, and C. Maihofer. Providing Reliable Agents for Electronic Commerce. *IEEE*, 1998.

[15] T. Walsh, N. Paciorek, and D. Wong. Security and Reliability in Concordia. In *Hawaii International Conference on System Sciences*, June 1998.

[16] Y.-M. Wang, Y. Huang, and W. K. Fuchs. Progressive Retry for Software Error Recovery in Distributed Systems. In *Proceedings of the International Symposium on Fault Tolerant Computing*, 1993.

[17] C. D. Wolfson, E. M. Voorhees, and M. M. Flatley. Intelligent Routers. In *9th International Conference on Distributed Systems*, June 1989.