An Application-Transparent, Platform-Independent Approach to Rollback-Recovery for Mobile-Agent Systems

Eugene Gendelman
Lubomir F. Bic
Michael B. Dillencourt

Department of Information and Computer Science
University of California
Irvine, CA 92717-3425 USA
Email:{egendelm, bic, dillenco}@ics.uci.edu

Abstract

This paper proposes a new approach to rollback-recovery for mobile-agent systems, and describes its implementation in the MESSENGERS mobile agents system. The used checkpointing method allows to implement space and time efficient, user-transparent rollback-recovery in heterogeneous distributed environments. Together with an efficient non-blocking system snapshot algorithm this checkpointing method is an attractive choice for implementing a rollback-recovery mechanism in the mobile agent system, because it exploits features specific to mobile agent systems during the recovery.

This paper also presents an optimization technique, called concurrent checkpointing, that increases the effectiveness of the proposed rollback-recovery mechanism.

1. Introduction

Distributed systems consisting of a network of workstations or personal computers are an attractive way to speed up large computations. These systems have a much higher performance-to-price ratio than large parallel computers, and they are also more widely available.

The major drawback is that the computing nodes in a distributed system or their connections may fail. As some applications may require hours to execute, it is important to be able to continue computation in the presence of node or link failures. Recovery from failures becomes more important for large systems, since the possibility of a failure increases with the number of computing nodes/links. Link failures can be solved by fault-tolerant communication protocols, and thus we only need to deal with failures of nodes.

Failure recovery may be achieved with a rollback-recovery mechanism. A rollback-recovery mechanism consists of three parts: checkpointing, fault detection, and failure recovery. During checkpointing the states of the participating processes are periodically saved on a stable storage. The saved process state is called a *checkpoint*. When a node failure occurs, the recovery mechanism uses saved checkpoints to recover the system to the consistent system state and continue execution from that state. The number of processes that have to be rolled back to the previous checkpoint varies, depending on the recovery algorithm. It may be necessary for one [1], some [2], or all [3] processes to roll back to the previous checkpoint.

It is useful for the checkpointing algorithm to take a checkpoint of each process such that the set of the local checkpoints represents a consistent system state, also called a *consistent system snapshot*. This facilitates fast recovery, since a consistent system snapshot does not have to be derived from uncoordinated local checkpoints. It also simplifies memory management, since as a new consistent snapshot is taken, the previous one can be discarded. In addition, a consistent system snapshot can be used for debugging purposes.

It is not always possible to restart the failed process on the same machine it ran before the failure. To allow recovery in a heterogeneous system, the process checkpoint must be architecture-independent. Another desirable feature of checkpoints in heterogeneous system is a portability of checkpointing routines. For example, computers are being purchased incrementally and added to the distributed system. As computer technology advances, new computers are available with every new purchase. Difficulties with porting of checkpointing routines can limit the availability of resources that could participate in distributed computations. Therefore, it is not sufficient for the checkpoints to be architecture-independent. The checkpointing functions must also be easily portable.

One way to implement a distributed system is using the mobile agent paradigm. In such systems, computation is performed by cooperating agents, which migrate through the system according to their individual programs. Most of the existing mobile agent systems are focusing on internet-based computing on LANs or clusters of workstations. However, mobile agent systems are also well suited for general-purpose computing. Applications include Monte-Carlo simulations, matrix multiplication, and individual-based simulations [4, 29]. As was shown in [4] mobile agent systems can show comparable performance with message-passing systems.

Interestingly, the architecture of the mobile-agent systems offers certain benefits in collecting a checkpoint and during the recovery. To demonstrate our approach, we describe the implementation of the rollback-recovery mechanism in the MESSENGERS mobile agent system. The rest of the paper is organized as follows: section 2 gives an overview of the MESSENGERS system, section 3 describes how the checkpoint of the individual process is taken, and section 4 shows the performance of the checkpointing method in section 3. Section 5 describes the algorithm for taking a consistent system snapshot. Section 6 presents a technique that optimizes the efficiency of the proposed checkpointing scheme. Section 7 describes the recovery algorithm and process merging: a feature of the recovery procedure specific to the mobile agent systems, section 8 presents related work and section 9 concludes the paper.

2. MESSENGERS system

MESSENGERS is a distributed system based on the principles of autonomous objects, called Messengers. MESSENGERS distinguishes three separate levels of network (figure 1). The physical network is the underlying computational resource. The daemon network is a collection of server processes, whose task is to interpret the behavior of Messengers, and system commands. Examples of possible system commands are initiation of the checkpointing, daemon failure notification, load balancing messages, and injection of a new Messenger. There is one daemon per physical node. The logical network is an application-specific computation network created at run time on top of the daemon network. Multiple logical network nodes may be created on the same daemon network node, which are then running on the same physical node. Logical nodes may be interconnected by logical links into an arbitrary topology.

Each link of the logical network has a name and several (optional) weights, which Messengers use for determining which links to route themselves along. Each logical node has a name and provides a memory space commonly accessed by all Messengers that gather on the node. This memory space, the Node Variable Area, functions as both a node-unique database and as an inter-Messengers communication channel.

Each Messenger can access three types of variables: messenger, node, and environmental variables. Messenger variables are local to and carried by the Messenger as it propagates through the logical network. Node variables are node-resident and are mapped to the Node Variable Area where the Messenger is currently running. There they are shared among Messengers running on the same node. Environmental variables provide information such as the current node name, and the name and weights of the last traversed link.

Messengers navigate through the logical network based on their own internal program and state. This is accomplished

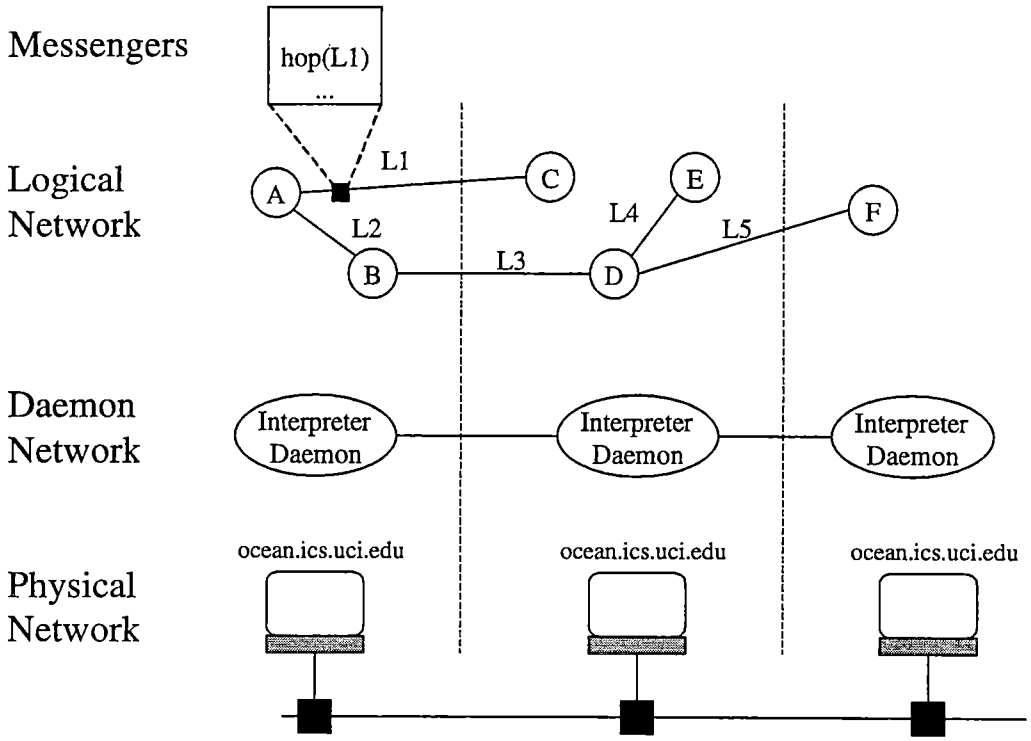


Figure 1

by explicit navigational statements, which also permit the creation or destruction of logical links and/or nodes. Messengers may also perform arbitrary computations in the nodes they visit. This can take two forms. First, the Messenger program may contain computational statements, and second, Messengers may invoke ordinary C functions. More information on MESSENGERS can be found in [5].

3. Capturing daemon state

When a Messenger is running, the daemon state is as complex as with any other program. But when no Messenger is running, the state is very simple: it consists of the data space shared by the Messengers (the logical network) and the collection of Messengers waiting for execution or sitting in input/output queues. Each Messenger is represented by a simple data structure, called the Messenger Control Block. Moreover, when the agent arrives to the daemon or prepares to migrate, its state is already captured in such a data structure.

Techniques to capture an agent state have been implemented in Odyssey [6], IBM Aglets [7], Ara [8], Agent TCL [9], Tacoma [10] and other mobile-agent systems. All these systems are interpreter-based, which makes them significantly slower than systems running compiled code. In MESSENGERS, agents are fully compiled into machine code, which makes the performance of MESSENGERS compatible with compiled message passing distributed systems.

A technique of compiling Messengers into native code that makes it easy to capture their state was first proposed in [4], and further explored in [28]. The Messenger script is first compiled by a special compiler that groups parts of the script into function blocks that are separated by migration statements and can be compiled with a standard compiler. An example of such precompilation is shown in figures 2, 3, and 4.

After a Messenger is compiled, it consists of a set of local variables, an array of compiled functions, and an index of the next function that must be executed. This index eliminates the need for saving a program counter. Moreover, in between functions, the stack does not contain any Messenger data. The heap in this approach is either disallowed, or private to the agent.

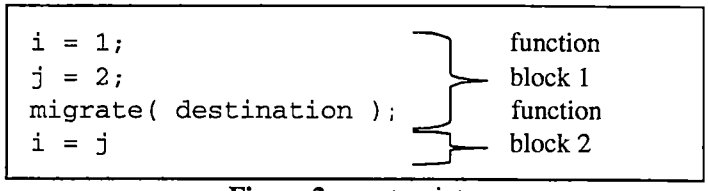


Figure 2: agent script

```
functionBlock_1(t_agent *a)
{
  a->agentVariables.i = 1;
  a->agentVariables.j = 2;

  a->nextFunctionBlock = 2;

    /*(code for migrate */
}
```

Figure 3: function block 1

```
functionBlock_2(t_agent *a)
{
  a->agentVariables.i =
     a->agentVariables.j;

  a->nextFunctionBlock = 0;
 /* function 0 indicates */
 /* end of agent script  */
}
```

Figure 4: function block 2

The Messenger code is compiled into a run-time library for every participating machine type in the system. When a Messenger migrates to a machine, it loads the library, (if it is not loaded yet) and calls the function specified by the index of the next function it carried along.

The main benefit of this approach is that the state of the daemon is captured in a data files, which is both machine and application independent. At the beginning of the file the type of the byte order used by the original host should be specified, so that the data is correctly loaded on machines that use alternative byte order convention. Since there are no system calls involved in taking a checkpoint, this checkpointing mechanism is portable to any architecture without modifications.

4. Performance evaluation of daemon state capture

This type of checkpointing also takes less space, and therefore less time to collect, than a regular process core dump. We did two experiments that qualitatively show the advantages of this checkpointing method. In the first experiment we implemented a synthetic application consisting of logical nodes and Messengers hopping around in the logical network. Since we are only concerned with the size of the checkpoints, the semantics of the application is irrelevant. The application was ran on a single Sparc-5 workstation.

We measured the size of checkpoints taken by Clubs, a version of libckpt [11], and by MESSENGERS. Figure 5 illustrates our results. The first column specifies number of logical nodes (N) and Messengers (M) participating in the computation. The second column shows the checkpointing size (in bytes) for Clubs, and the third for MESSENGERS.

	Clubs	MESSENGERS
1 N, 1 M	1665768	1684
10 N, 10 M	1669864	12360
20 N, 20 M	1686248	22096
30 N, 30 M	1702632	39320

Figure 5

The difference in the size of the checkpoint comes from the fact that Clubs saves the state of the daemon process, including all daemon

variables, which MESSENGERS saves only the logical network and the individual Messengers. After the first row, in both Clubs and MESSENGERS, the size of checkpoints grows linearly with addition of new logical nodes and Messengers.

In the second experiment the Messenger script was modified to call a C function that initializes a thousand-by-thousand array of characters. Figure 6 presents the results of this experiment. Comparing with the results of figure 5 the size of the checkpoint in Clubs increased by 1,000,000 bytes, since it includes an array of [1000][1000] characters into the checkpoint. The size of the MESSENGERS checkpoint did not change. This is because in our checkpointing mechanism checkpoints are taken only at the script level, i.e, *between* Messengers functions, and therefore, local variables of the C function do not have to be saved. The overhead from taking the checkpoints within functions also grows with the number of execution threads. The current implementation of MESSENGERS has only one execution thread per processor.

	Clubs	MESSENGERS
1 N, 1 M	2665192	1684
10 N, 10 M	2672360	12360
20 N, 20 M	2681576	22096
30 N, 30 M	2714344	39320

Figure 6

In these experiments we did not consider possible optimizations to the Clubs, such as incremental checkpointing [16]. With incremental checkpointing only memory pages that were modified since the time of the previous checkpointing are saved to the checkpoint, which makes the checkpoint size smaller. A similar technique could be applied in our approach to unmodified shared memory areas, and unmodified Messengers.

5. Consistent system snapshot algorithm

In this section we describe the algorithm to capture a consistent system snapshot. The idea is to collect a set of checkpoints that represent a consistent system state. Section 5.1 gives a definition of a consistent system state. Section 5.2 describes the algorithm that was used in the MESSENGERS system, followed by MESSENGERS-specific parts of the algorithm. In section 5.3 we describe how logical network is created and modified in MESSENGERS, and in section 5.4 we present the complete algorithm.

5.1. Consistent system state

The definition of a consistent system state varies, depending on the underlying communication channels. *Reliable* communication channels guarantee that every message that was sent will also be delivered to its destination. *Unreliable* communication channels dont provide such a guarantee. For systems using unreliable communication channels *a consistent system state is one in which every message that has been received is also shown to have been sent in the state of the sender* [13]. In distributed systems using reliable communication channels a consistent system state also includes in-transit messages since they will always be delivered to their destination in any legal execution of the program [2]. Hence, we add to the above definition that *a consistent system state is one in which every message that has been sent is also shown to be received in the state of the receiver.*

Figure 7 shows the types of inconsistencies that checkpointing algorithms should avoid in order to take a consistent system snapshot. In these diagrams, checkpointing is represented by a dashed line. The checkpointing line is not strait because clocks of different computing nodes are not synchronized. Instead checkpointing is synchronized by message exchanges, and message transmission in asynchronous systems can take arbitrary time. When the system snapshot line crosses the process line, the checkpoint of the process is taken.

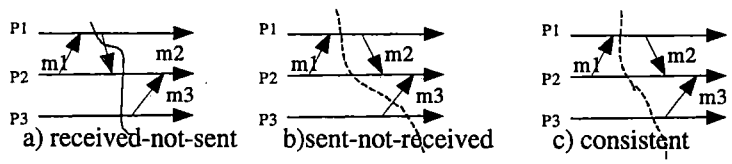


Figure 7

When a snapshot is taken as in figure 7a, message m2 will be registered as received by P2, but will not be registered as sent by P1. This is called a *received-not-sent* inconsistency. Figure 7b illustrates a *sent-not-received* inconsistency; m3 is registered as sent by P3, but it is not registered as received by P2. This type of inconsistency applies only to systems with reliable communication channels, since in systems that use unreliable channels message loss is allowed. The snapshot shown in figure 7c is an example of a consistent system snapshot.

Reliable communication channels are harder to provide than unreliable channels. The guaranteed message delivery may be provided by the hardware, by communication protocol, or by a user program. Supporting reliable communication channels places a more stringent requirement on the checkpointing algorithm. If the checkpoints are taken on the level relying on the guaranteed message delivery, then the checkpointing algorithm must guarantee that all the messages that were sent were also received.

The two main requirements for the algorithm that captures a consistent system snapshot in our case are 1) the snapshot algorithm should implement reliable communication channels. This allows using standard communication protocols that provide guaranteed message delivery, such as TCP/IP, for inter-process communication. 2) The algorithm should be non-blocking, since checkpoints are taken only in between agent functions.

Another concern for the snapshot algorithm is the number of required communication messages. Most algorithms require $O(n^2)$ communication messages, where n is the number of processors in the system. This makes them unnecessarily slow as the number of participating nodes grows.

5.2. Checkpoint Algorithm

One distinguished checkpoint server acts as a checkpoint coordinator. Each process maintains one permanent checkpoint, belonging to the most recent consistent checkpoint. During each run of the protocol, each process takes a tentative checkpoint, which replaces the permanent one only if the protocol terminates successfully. Each checkpoint is identified by a monotonically increasing Checkpoint Number (CN). Every application message is tagged with the CN of its sender, enabling the protocol to run in the presence of message re-ordering or loss.

Each process counts the number of messages it sent and received in a *srDelta* (sent/received delta) variable. Every time a message is sent, the srDelta is incremented. When a message is received, srDelta is decremented. The resulting algorithm is presented below.

1. The coordinator process starts a new consistent checkpoint by taking a tentative checkpoint, incrementing CN, and broadcasting an *Initiate* message containing CN.
2. Upon receiving an *Initiate* message, a process takes a tentative checkpoint, and increments its local CN. It sends a *cpTaken* message to the coordinator, including its srDelta.

   If a process receives an application message with CN greater than its own, it also takes a checkpoint before processing the message.
3. When the coordinator *(root)* process receives a *cpTaken* message from all processes *(its children)*, and if its srDelta is zero, coordinator *(root)* broadcasts a *Commit* message *(to its children)*.
4. Handling late messages: when a message arrives whose CN is less than the current CN, a copy of the message is appended to the log file of the checkpoint identified by the message CN. An *Update* message is sent to the coordinator.
5. When the coordinator process receives an *Update* message, it increments its srDelta. If srDelta equals zero, the coordinator broadcasts a *Commit* message.

   Steps 4 and 5 are repeated until all messages that were sent with a given CN are also received.
6. When a process receives a *Commit* message, it makes its tentative checkpoint permanent and discards its previous permanent checkpoint.

The important characteristics of this algorithm are that it is non-blocking, has a minimum storage requirements (only one complete system snapshot is stored at all times). Its message complexity is $O(n + m)$, where n is the number of participating processes, and m is a number of late messages. Another important characteristic of this algorithm is that it implements reliable communication channels. This is necessary for rollback-recovery in the systems relying on guaranteed message delivery by the underlying communication layer, such as TCP/IP.

5.3. Modifying the logical network in MESSENGERS

An example of a logical network is shown in figure 8. Each link, although presented as a single structure to the user, consists of two parts, one for each direction. If a logical link spans two daemons, such as L1, the information is repeated on both daemons. Each link has a daemon-unique id. Every link also knows the id of its counterpart, and the daemon address where the destination node resides. The link pointer is stored in two data structures. One data structure is accessible from the logical node. It is the list of pointers to all the links connected to the logical node. Another data structure is a tree, accessible from the daemon. This tree contains the pointers to all the links residing on the daemon, sorted by the link id.

This is how these structures are used to accommodate Messenger transfer from one node to the other. Messenger finds the link it wants to traverse from the link list of its current logical node. From the link information Messenger knows on what physical machine the destination node resides, and the id of the link on the destination daemon. Messenger is transferred to the destination daemon, searches the link by the link id in the daemon link tree. From the link information Messenger acquires the pointer to the destination logical node.

Lets look at how Messenger builds a logical network. Creation of the logical link can be broken into three steps as illustrated on figure 9a. In the first step a link data structure is created on the first daemon and daemon-unique link id is

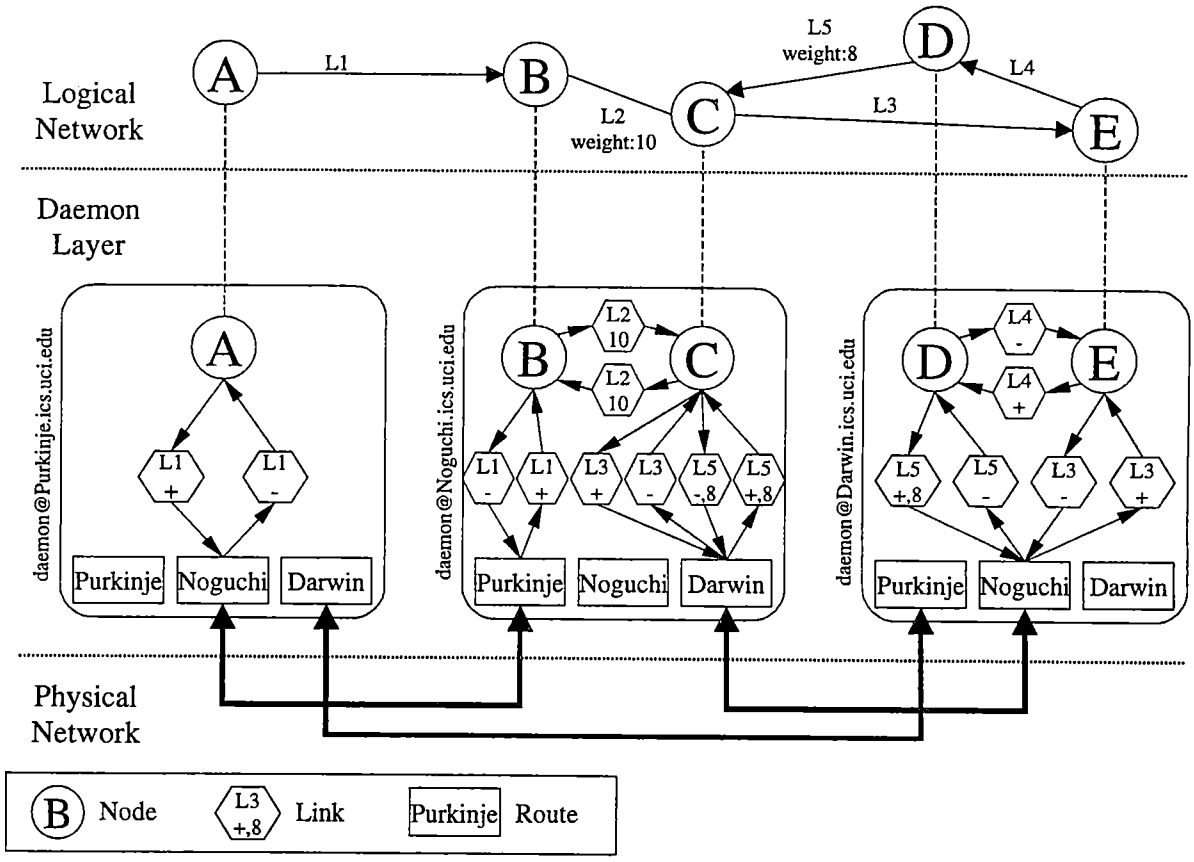


Figure 8

assigned to it. New link is registered in the daemon link tree, but not in the node a link list. After this operation link can be followed to node a, but not from it, as indicated by the arrow. A star above the link indicates on which daemon changes are made in this step.

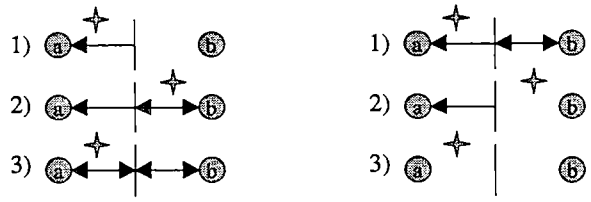


Figure 9

In step two a Messenger is transferred to the destination daemon, carrying id of the link created in step 1. Messenger creates a new link with a daemon-unique link id. This link is registered both in the daemon link tree and node b link list. The value of the link id created in the step 1 is assigned to the destination link id field of the new link. After this operation, link can be traversed from node b to node a.

In the third step a message is sent to the daemon, host of the node a, carrying the id of the link created in the step 2. As the destination link id field of the first link is updated, the link is added to the node link list so that Messengers can traverse this link from node a to node b.

Delete operation is also done in three steps, as illustrated in figure 9b. In the first step a link is disconnected from the logical node, so that Messengers can not traverse it from a to b. In the second step the second link is completely destroyed. In the third step a message is sent to the host of a that destroys the first link.

5.4. Handling logical network modifications in checkpointing

As create and delete operations are not atomic; they might not be completed in a single checkpoint. This may lead to a system with inconsistent logical network.

Example 1: in figure 9a node a is created, node b is created, and checkpoint is taken. CN is incremented. Message is sent to the first daemon to create the link from a to b. Since message comes with increased CN, it will force the first daemon to take a checkpoint *before* processing the message, and the link from a to b will not be included in the checkpoint.

This can be avoided by not incrementing the CN of the message sent from b to a. This, however, introduces another problem. In the checkpointing algorithm there is a sent-received delta (srDelta) counter. When a message is sent out, srDelta is incremented, when the message is received srDelta is decremented. Then the srDeltas of all processes added together, and if the sum equals to zero, checkpoint is committed. Since now messages with the old CN can be sent even after checkpoint is taken, there is no way to say when checkpoint can be committed, as illustrated in the following example.

Example 2: in figure 9a node a is created, node b is created, and checkpoint is committed before step 3 is made. One computer fails and the system is restarted from the last committed checkpoint, which does not have a link from a to b.

The checkpoint is not committed unless the number of the reported sent messages equals to the number of the reported received messages. We can use this fact to ensure that creating and deleting logical links and nodes is finished in the same checkpoint where it was started. The message leaving the daemon to create a new link or a new logical node increments srDelta not by one, but by two. When it is received by the other daemon, srDelta is decremented only by one. The last count is saved for the update message sent in step 3. This message does not increment srDelta of the sender daemon, but decrements the srDelta of the receiving daemon. This scheme guarantees that link creation is started and finished in the same checkpoint interval.

The only logical links that are saved in checkpoints are the ones connected to the logical nodes. This means that the delete link operation does not cross with the checkpointing on the first daemon, since link is disconnected from the logical node in the first step of the protocol, and therefore it is not saved. This means that message sent from b to a does not affect the saved logical network, and does not need to have any particular CN. Because of this fact we dont need a special counting in delete operation; each Messenger counts as one on sending and receiving daemons, and all other messages are ignored in counting.

Messengers have the ability to modify the logical network during the system execution. There are three ways Messengers can modify the logical network. Messengers can create logical links and logical nodes. Messengers can destroy logical links and nodes, and Messengers can modify logical link and node information, such as link names and weights. Because Messengers can arrive after a checkpoint is taken, these changes need to be reflected in the checkpoint. When such a Messenger that modifies logical network arrives, the appropriate information is appended to the checkpoint. This information includes the id of the node or link, and operation code that needs to be performed. These operations include add node, add link, remove node, remove link, modify node, and modify link. Figure 10 summarizes how create and delete operations are handled. Last column specifies what additional information is appended to a checkpoint.

step	send	count rcvd	late message
1)	2	0	
2)	0	1	add node and/or
3)	0	1	add link

Create operation

step	send	count rcvd	late message
1)	1	0	
2)	0	1	delete node and/or
3)	0	0	

Delete operation

Figure 10

6. Concurrent Checkpointing

With many benefits of taking checkpoints in between Messengers functions there is a drawback: if an agent is executing a long function, checkpointing will not proceed to the next stage until the function is completed. This does not cause any additional delay to the main computation, because a non-blocking snapshot algorithm is being used, but it prolongs the time needed for checkpointing to commit. This limits the frequency with which system snapshots can be taken, reducing the usability of the rollback-recovery mechanism.

To deal with this problem, the checkpointing protocol can be made *concurrent*: multiple checkpointing protocols can be active at the same time. Concurrent checkpointing does not speed up the time it takes for the individual checkpointing protocol to complete, but it removes the limitation on how often checkpoints are initiated. In the following subsections we present a mechanism implementing concurrent checkpointing with the algorithm presented in section 5. Section 6.1. and 6.2. present the implementation of concurrent checkpointing. Section 6.3. presents a proof of correctness, and section 6.4. discusses the benefits of concurrent checkpointing in more detail.

6.1. Checkpointing States

Each process (daemon) maintains a tree of checkpoint structures shown in figure 11, one for each checkpoint in progress. As soon as the system execution starts, the first `s_checkpoint` structure is created. There are four distinct states of the checkpointing protocol that require different actions. When `s_checkpoint` structure is created, the checkpointing protocol is in the *Initial* state. In this state the work of the checkpointing mechanism is to count the

```
struct s_checkpoint{
  int     CN;        // Checkpoint Number
  int     srDelta;   // Sent/Received delta counter
  int     state;     // Initial, In Progress, Complete, Committed
};
```

Figure 11

difference between sent and received messages. When the checkpointing protocol is initiated and daemon takes a checkpoint, the state of the checkpointing protocol is updated to *In Progress.* In this state, a process increments its CN and appends all the arriving messages with the CN smaller than the current process CN to the checkpoint, as described in the next section. When the coordinator announces that the system sum of the srDeltas is zero, the status of checkpointing is updated to *Complete.* If the state of the previous checkpointing is *Committed,* then the state of the

current checkpointing is also upgraded to *Committed* and the previous `s_checkpoint` structure is removed from the checkpoint tree, and the corresponding checkpoint is removed from the stable storage. When the checkpoint state is upgraded to *Committed*, the daemon checks whether the checkpoint with a higher checkpoint number is *Complete* and is waiting to be *Committed*. If so, the state of that checkpoint is also upgraded to *Committed*, and so on. Figure 12 summarizes the actions taken in different checkpointing states. The differentiation between the *Complete* and *Committed* states guarantees that a checkpoint is committed only after all the previous checkpoints are committed.

State	Action
Initial	Update SRC
In Progress	Update SRC, add new info to checkpoint file
Complete	Wait until all previous checkpoints are committed
Committed	Remove earlier checkpoint structures and files

Figure 12

6.2. Including late messages in checkpoints

When a message with CN less than the current process CN, also called a *late* message, arrives, it must be appended to the corresponding checkpoints. Section 5.1 describes how this is done in the case when a single checkpointing algorithm is in progress at any time. Here we address this issue in the case of concurrent checkpointing. The rule to decide whether to append new information to the checkpoint or not is

> *Message with CN = **a** has to be appended to all checkpoints **x** of the process with current CN = **b**, such that **a** <= **x** < **b**.*

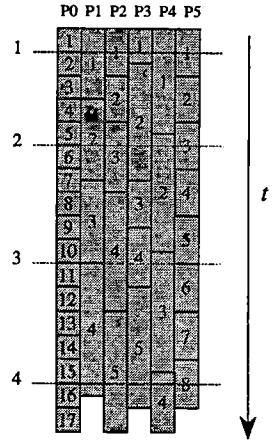
The following example provides an intuitive explanation of this solution: a message *M* with CN = 5 arrives at the process with a current CN = 7. Message *M* needs to be included in checkpoint 5. It also needs to be included in checkpoint 6. This is because the arrival of *M* is not recorded in checkpoint 6, since it was taken before the message arrival.

6.3. Proof of correctness

Definition 1 A checkpointing protocol captures a consistent system state if for any checkpoint taken, all received-not-sent and sent-not-received inconsistencies are eliminated.
Assumption Serial snapshot algorithm captures a consistent system state.
Theorem 1 *Presented algorithm for concurrent checkpointing captures a consistent system state.*
Proof The received-not-sent inconsistencies are eliminated, since if a message with higher CN arrives, the checkpoint is taken before processing the message. The sent-not-received inconsistencies are eliminated since messages with lower CN are appended to the checkpoint and the checkpoint protocol is not complete until all the messages that were sent before the checkpoint was taken are also received. This follows since a checkpoint is not committed until all the previous checkpoints are also committed.

6.4. The impact of concurrent checkpointing

To demonstrate the impact of the concurrent checkpointing we created a synthetic application running on six machines so that agents executing on five machines (P1, P2, P3, P4, P5) invoke functions that are generally longer than the interval with which checkpointing is initiated. The sixth machine (P0) executes agents whose functions take exactly checkpointing interval to execute. For simplicity, we assume that P0 is a coordinator daemon that triggers the checkpointing. We use only one such daemon because any other daemons executing short agent functions would behave exactly the same as P0. For this example we also disregard network latency.

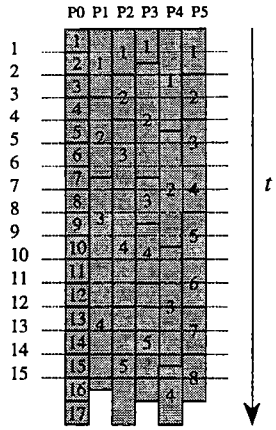


CP #	P0	P1	P2	P3	P4	P5
1	1	1	1	1	1	1
2	5	2	3	2	2	3
3	10	3	4	4	3	5
4	15	4	5	5	4	8

Figure 13 shows checkpoints taken with the serial scheme.

Figure 13. Non - concurrent checkpointing

The rectangles represent agent functions. The time line goes down so the agent functions executed on the same processor are in the same column. Dashed lines represent checkpoints. Even though checkpoints can be initiated after each task on the processor P0, there are only four checkpoints taken. After the first checkpoint is triggered, the checkpoint can not proceed until all the processes see the checkpoint request, which means they have to be out of the agent functions. As soon as the last daemon replies, P4 in this case, checkpoint is committed and a new one can be initiated. The table in figure 13 shows the agent functions that were included in each checkpoint on every daemon.



CP #	P0	P1	P2	P3	P4	P5
1	1	1	1	1	1	1
2	2	1	1	2	1	1
3	3	1	2	2	1	2
4	4	2	2	2	1	2
5	5	2	3	2	2	3
6	6	2	3	2	2	3
7	7	3	3	3	2	4
8	8	3	4	3	2	4
9	9	3	4	4	3	5
10	10	4	4	4	3	5
11	11	4	4	5	3	6
12	12	4	4	5	3	6
13	13	4	5	5	3	7
14	14	4	5	5	3	7
15	15	4	5	5	3	8

Figure 14. Concurrent checkpointing

Figure 14 illustrates what happens in the case of concurrent checkpointing, when the next snapshot algorithm can be initiated independently of all the previous runs of the algorithm. Concurrent checkpointing allows to collect more fine-grained snapshots than non-concurrent checkpointing. For example, checkpoint #2 in the serial case corresponds to checkpoint #5 in the case of concurrent checkpointing. Checkpoints #2, #3, and #4 in the concurrent checkpointing are missing in the serial scheme.

Note, that the checkpoints of the processes that did not change since the previous checkpoint need not be saved again.

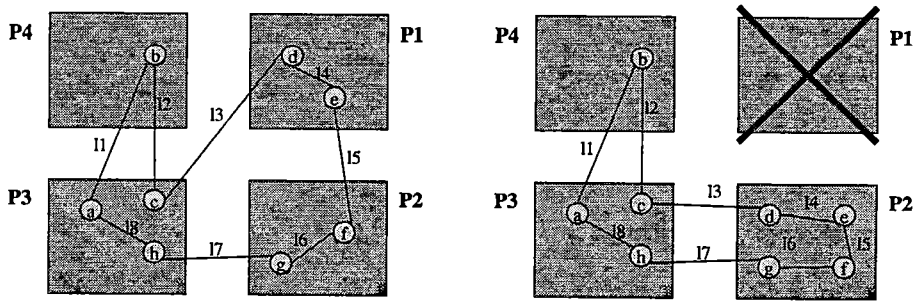
7. Failure recovery

In this section we discuss issues of failure recovery. In section 7.1. we describe process merging, a feature specific to the mobile agent systems. Section 7.2. discusses how a single point of failure is handled. Section 7.3. shows the failure recovery algorithm supporting process merging.

7.1. Process merging

The state of the MESSENGERS daemon from the application viewpoint consists of the set of the logical nodes, logical links, connecting these nodes, and mobile agents, doing a computation in this logical environment. Since the logical network, from the application viewpoint, is completely independent of the physical network, it is possible to merge the checkpoint of the failed process into the checkpoint of the coordinator as shown in figure 15.

Here the daemon processes are identified by rectangles, logical nodes as circles and logical links are the lines connecting these circles. After the daemon on P1 fails, the logical nodes *d* and *e* are moved to P2.



a) system before the failure

b) system after the failure

Figure 15

Merging allows to have only one daemon per physical node at all times. This means that an ip address is sufficient to uniquely identify the daemon process in the system. Having one process per physical node facilitates simpler and more efficient load balancing, since balancing between processes on the same machine need not be considered. Process merging also helps to eliminate unnecessary interprocess communications between daemons residing on the same physical node.

Merging, however, can introduce imbalance into the distributed system. As shown in figure 15, after process P1 is merged into P2, P2 hosts half of the logical nodes of the system. However, logical nodes can be offloaded from the daemon to other daemons using a similar mechanism as that used for merging. A load balancing mechanism was developed that can move logical nodes around the system, minimizing the systems idle time. Together with this load balancing mechanism daemon merging can be used efficiently.

In our implementation a failed daemon is always merged with the coordinator. Even though more sophisticated strategies could be used to reduce the imbalance introduced by the failure, we choose this simple rule for its simplicity and let the load balancing algorithm do the balancing.

7.2. Single point of failure

Both our checkpointing algorithm and recovery algorithm rely on a central coordinator. Therefore the question of coordinator failure has to be considered. A simple rule to elect a new coordinator is to elect the one with the smallest IP address. Each daemon has the list of all participating processes. When a daemon receives a notification that another daemon failed, it removes it from its daemon list and selects a new coordinator independently of other daemons. Since all daemons have information about all other daemons in the system, they all will select the same new coordinator. (We did not consider the case when the distributed system is split into two parts, since it is not applicable to computing on a LAN.) This election scheme does not require the exchange of extra messages, other than notification of failure, which is done anyway in the course of recovery.

7.3. Recovery algorithm

We assume that if a process fails, one or more processes will detect this failure. Failure detection is based on the quality of the TCP/IP protocol to signal the sender when receiver is not reachable. Our recovery protocol assumes that communication channels are reliable and obey FIFO order, which is also supported by TCP/IP.

The recovery protocol consists of two parts. In the first part a new set of live processes is determined, and all the messages that were sent before the failure are eliminated from the system. The basic idea is that all the processes send a message to all the other live processes in the system. After every pair of live processes exchange such messages, the communication channels are clear of messages sent before the failure, and all participating processes know about all live processes in the system. In the second part of the protocol the system is restarted from the previous checkpoint. The algorithm proceeds as follows.

1. When a process is notified of unknown failure it reinitializes the daemon and broadcasts the *Failure* message that includes the ip of the failed daemon and checkpoint number (CN) that needs to be loaded.
2. When a process receives the *Failure* message from all other processes in the system, it sends a *Recover* message to the coordinator process, and loads the checkpoint.
3. When coordinator receives *Recover* message from all live processes, it loads the checkpoint of the failed daemon, appending it to its own checkpoint, and sends a *LinkUpdate* message to corresponding daemons to update failed links.
4. Daemons that receive *LinkUpdate* message update corresponding links and send *LinkUpdateAck* message to coordinator.
5. When coordinator receives all *LinkUpdateAck* messages, it broadcasts an *Activate* message.
6. After receiving *Activate* message, daemon resumes computation.

In the example on figure 15 the coordinator P2 in the third step of the protocol will send a message to P3 to update the destination of the link l3. Links l4 and l5 will be updated locally on P2.

This algorithm works correctly even in the presence of multiple failures. After every new failure the algorithm restarts by flushing all communication channels, thereby discarding all previous messages.

8. Related Work

In recent years several projects have considered checkpointing in heterogeneous systems. A number of different approaches have been proposed. The Dome distributed system [19] uses the SPMD model for application parallelization. The program is organized as a loop. In their approach checkpoints are taken automatically at the beginning of the loop. This avoids saving the program counter and the stack. Another method implemented in Dome is to use a preprocessor to insert extra statements into the code, to save the program counter and the stack. This approach has some limitations: the user is responsible for saving C++ variables that are in the scope at the time of a checkpoint.

A checkpointing library was developed by Silva, Veer, and Silva [20]. In their approach the user is responsible for specifying the data that has to be included into checkpoints. The resulting checkpoints are architecture-independent.

Another approach to checkpointing was implemented in the Charm++ system [21]. Charm++ is an object oriented parallel language based on C++. It uses message passing between objects, called *chairs*. In Charm++ checkpoints are taken only when none of the chairs are executing functions, which allows capturing the state of the chair as a set of chair variables and a list of messages that arrived at the chair. In this respect, their approach is similar to the one proposed in this paper. However, they used a blocking algorithm to collect a consistent system snapshot.

Ramkumar and Strumpen [22] proposed a source-to-source compilation to support portability. A C program is pre-compiled with the c2ftc compiler. The resulting program is instrumented with checkpointing and recovery code. That program is compiled with a regular C compiler. The prototype presented in [22] is efficiently used for providing fault tolerance to a single-process application.

A compiler-based approach for process migration was proposed by Theimer and Hayes [23]. When a process is migrating, its state is translated into a machine-independent state. The migratory program that represents that state is generated and compiled on the target machine. When this migratory program is run, the process is recreated. Implementation of this approach was not presented.

The table in figure 16 summarizes the characteristics of different approaches to checkpointing. Each approach uses different methods to optimize the size of checkpoints. The application-level checkpoints (whether supported by the run-time library or not) can potentially capture the checkpoint most efficiently, since they have the most knowledge of the application program. The price for this is application transparency.

Checkpoint type	Porta-bility	Trans-Parency	Check-point Porta-bility	Application Support	System Support	Consistent system state	Examples
Application-based	Yes	No	Yes	All the work	None	Captured by application	[24]
Run-time library	Yes	No	Yes	When and what to checkpoint	Checkpointing routines	Ensured by checkpoint placement	[20,25,26]
Compiler-based	Yes	Almost	Yes	When to checkpoint	None	Not implemented	[19,22,23, 27, 28]
Environment-based	Yes	Yes	Yes	Optional	Checkpointing supported by the environment	Captured by the environment	[19,21], this work
System-based	No	Yes	No	Optional	All	Not implemented	[3,11]

Figure 16

There has been much research in designing checkpointing algorithms [2, 3, 13-18]. However, none of these algorithms satisfy all our requirements: algorithms [13, 15, 16] do not implement reliable communication channels. Algorithms presented in [2, 3, 15] are blocking. Algorithm [17] relies on the assumption that all the processor clocks are approximately synchronized, which limits the generality of these algorithms.

Several non-blocking algorithms [14, 18] require less than $O(n^2)$ communication messages. The Kai Li algorithm [14] performs well on multicomputers. It requires $O(nlogn)$ messages for hypercube connected multicomputers and $O(n)$ for mesh connected multicomputers. However, this algorithm depends on the knowledge of the process interconnection topology, which is unusable in systems where the pattern by which processors are connected varies, as in systems constructed by interconnecting PCs or workstations. Furthermore, this algorithm requires communication channels to be FIFO. The Silva algorithm [18] requires only $O(n)$ communication messages. However, it relies on the knowledge of fault detection latency, and message latency, which might be difficult to determine.

9. Conclusion

It was shown in [4] that mobile agent systems can be effectively used to solve general computing problems. In this paper we investigated the rollback-recovery mechanism for systems based on the mobile agent paradigm. This rollback-recovery mechanism can be applied in the heterogeneous environment and requires minimum modifications for porting. Modifications are required for data marshalling between different platforms.

The proposed approach to checkpointing a single daemon process is general for the mobile agent systems, and it captures the state efficiently. It allows to capture the state of the process transparently to the user, and save a checkpoint in a machine independent format, so that the process state could be reloaded on a machine with a different architecture. The drawback of this approach is checkpointing routines are sensitive to the modifications of the distributed system itself. Another drawback is that the user does not have full control of the exact time when the checkpoint will be taken. A non-blocking snapshot algorithm and a concurrent checkpointing technique presented in this paper compensate for this drawback.

The daemon checkpoints consist of a set of the independent checkpoints of the individual logical nodes. This allows merging of a failed process with a running process instead of starting a failed process on a separate machine. Merging allows having only one daemon per physical node at all times. This facilitates simpler and more efficient load balancing, and eliminates unnecessary interprocess communications between daemons residing on the same physical node.

References

[1] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Trans. on Computers Special Issue On Fault Tolerant Computing*, 41(5):526-531, May 1992

[2] Yong Deng and E. K. Park. Checkpointing and Rollback-Recovery Algorithms in Distributed Systems. *Journal of Systems and Software*, 25:59-71, 1994

[3] J. Leon, A. L. Fisher, and P. Steenkiste. Fail-Safe PVM: A portable package for distributed programming with transparent recovery. *Tech. Rep. CMU-CS-93-124*, Carnegie Mellon Univ., February 1993

[4] Christian Wicke, Lubomir F. Bic, Michael B. Dillencourt, Munehiro Fukuda. Automatic State Capture of Self-Migrating Computations in Messengers. ICSE98 Intl Workshop on Computing and Communication in the Presence of Mobility, Kyoto, Japan, April 1998. http://www.ics.uci.edu/~bic/messengers/messengers.html

[5] MESSENGERS. http://www.ics.uci.edu/~bic/messengers/messengers.html

[6] Introduction to the Odyssey API. http://www.genmagic.com/technology/odyssey.html

[7] Bill Venners. Under the Hood: The architecture of aglets. JavaWorld, April 1997 http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html

[8] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In Kurt Rothermel, Radu Popescu-Zeletin, editors, Proc. of the First International Workshop on Mobile Agents MA'97. (Berlin, Germany), April 1997. http://www.uni-kl.de/AGNehmer/Projekte/Ara/index_e.html

[9] Robert Gray, George Cybenko, David Kotz, and Daniela Rus. Agent Tcl. In William Cockayne and Michael Zypa, editors, Itinerant Agents: Explanations and Examples with CDROM. Manning Publishing, 1997. ftp://ftp.cs.dartmouth.edu/pub/kotz/papers/gray:bookchap.ps.Z

[10] Johansen, D., van Renesse, R. and Schneider, F. B. An Introduction to the TACOMA Distributed System, Technical Report 95-23, Dept. of Computer Science, University of Tromco, Norway, 1995. http://www.cs.uit.no/Localt/Rapporter/Reports/9523.html

[11] J.S.Plank, M.Beck, G.Kingsley, K.Li. libckpt: Transparent Checkpointing Under UNUX, Conference proceedings USENIX Winter 1995 Technical Conference, January 1995.

[12] E. N. Elnozahy, D. B. Johnson, Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message Passing Systems, Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, October 1996

[13] K.M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Systems*, 3(1):63-75, February 1985

[14] Kai Li, Jeffrey F. Naughton, James S. Plank. Checkpointing Multicomputer Applications. In *Proc. of the 10th Symposium. on Reliable Distributed Systems,* pages 2-11, 1991

[15] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems *IEEE Trans. Software Eng.*, SE-13(1):23-31, January 1987

[16] E. L. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proc. of the 11th Symposium on Reliable Distributed Systems*, pages 39-47, October 1992

[17] F. Cristian, F.Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proc. of 10th Symposium on Reliable Distributed Systems*, pages 12-20, 1991

[18] L. M. Silva and J. G. Silva. Global checkpointing for distributed programs. *In Proc. of the 11th Symposium on Reliable Distributed Systems*, pages 155-162, 1992

[19] J. Arabe, A.Beguelin, B.Lowekamp, E.Seligman, M.Starkey, P.Stephan. Dome: Parallel Programming in Heterogeneous Multi-User Environment, Technical Report, Carnegie-Mellon University, April 1995

[20] Luis M. Silva, Joao G Silva. System-level versus User-Defined Checkpointing, Symposium on Fault-Tolerant Computing, pp 68-74, 1998

[21] Sanjeev Krishnan, Laxmikant V. Kale. Efficient, Language-Based Checkpointing for Massively Parallel Programs. Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign. Report #94-2. http://charm.cs.uiuc.edu

[22] B. Ramkumar, V.Strumpen. Portable Checkpointing for Heterogeneous Architectures, Proc. 27 th Fault-Tolerant Computing Symposium, FTCS-27, Seattle, June 1997

[23] M. M. Theimer, B. Hayes, Heterogeneous Process Migration by Recompilation, Proceedings of 11 th International Conference on Distributed Computing Systems, pp. 18-25, July 1991

[24] P.McGrath, B.Tangney. Scrabble A Distributed Application with an Emphasis on Continuity" Software Engineering Journal, pp. 160-164, May 1990

[25] Y. Huang, C. Kintala. Software Implemented Fault-Tolerance: Technolog ies and Experience, Proc. 23rd Fault-Tolerant Computing Symposium, FTCS-23, pp. 2-9, 1993

[26] Kuo-Feng Ssu, W. Kent Fuchs. PREACHES Portable Recovery and Checkpointing in Heterogeneous Systems, Proc. 28 th Fault-Tolerant Computing Symposium, FTCS-28, Munich, Germany, June 1998 http://computer.org/conferen/proceed/ftcs/8470/8470toc.htm

[27] Smith, P. and Hutchinson, N. Heterogeneous Process Migration: The Tui System, Software. Practice and Experience, 28(6), 611-639 (May 1998)

[28] Christian Wicke, Lubomir F. Bic, Michael B. Dillencourt. Compiling for Fast State Capture of Mobile Agents, Parallel Computing 99 (ParCo99). TU Delft, The Netherlands. August 1999

[29] L. F. Bic, M. Fukuda, M. B. Dillencourt, F. Merchant. MESSENGERS: Distributed Programming Using Mobile Autonomous Objects, Journal of Information Sciences, 1998 http://www.ics.uci.edu/~bic/messengers/prog.ps