Efficient algorithms for fault tolerant mobile agent execution

Jin Yang and Jiannong Cao*

Internet and Mobile Computing Lab,
Department of Computing,
Hong Kong Polytechnic University,
Hung Hom, Kowloon, Hong Kong
E-mail: csyangj@comp.polyu.edu.hk
E-mail: csjcao@comp.polyu.edu.hk

Weigang Wu
Department of Computer Science,
Sun Yat-sen University,
Guangzhou 510275, China
E-mail: wuweig@mail.sysu.edu.cn

Cheng-Zhong Xu
Department of Electrical and Computer Engg.,
Wayne State University,
Detroit, Michigan 48202, USA
E-mail: czxu@ece.eng.wayne.edu

Abstract: Redundancy is necessary for fault tolerance, but the overhead introduced by redundancy may degrade system’s performance. In this paper, we propose efficient replication-based algorithms for fault-tolerant mobile agent execution, which enable parallel processing in the agent execution to reduce the overhead caused by redundancy. We also investigate failure detection mechanisms and identify the problems of the heartbeat style failure detection approach and modify it for use in our proposed algorithms. Performance evaluation has been performed to compare the proposed algorithms with the existing algorithm. Both analytic and simulation results show that our new algorithms can significantly improve the system performance.

Keywords: fault tolerance; mobile agent; parallel processing; failure detector.


Biographical notes: Jin Yang is a PhD student at the Department of Computing, The Hong Kong Polytechnic University, Hong Kong. He worked for Lucent Technologies Optical Networks for three years after receiving his MSc from Xiamen University, China. His research interests include fault-tolerant distributed computing, mobile computing, failure detectors and agreement protocols.

Jiannong Cao is a Professor in the Department of Computing at Hong Kong Polytechnic University. He received his BSc in Computer Science from Nanjing University, China, in 1982, and his MSc and PhD from Washington State University, USA, in 1986 and 1990. He is Vice Chairman of the Technical Committee on Computer Architecture, an Executive Member of the Technical Committee on System Software, and a Senior Member of the China Computer Federation. He is a Senior Member of the IEEE, and a member of the ACM. His research interests include parallel and distributed computing, networking, mobile computing, fault tolerance, and distributed software architecture and programming.

Weigang Wu received the PhD Degree in computer science in 2007 from Hong Kong Polytechnic University, Hong Kong, where he has been a Postdoctoral Fellow. He received the BSc Degree in material science in 1998 and the MSc Degree in computer science in 2003, both from Xian Jiaotong University, China. He joined Sun Yat-sen University, China, in 2008. His research interests include distributed algorithm, fault tolerance, mobile computing and wireless networks.
1 Introduction

A Mobile Agent (MA) is a program that can migrate from host to host in a network of heterogeneous computer systems to execute the tasks specified by its owner. The migration path can be fixed according to a predefined itinerary or dynamically decided using a self-initiated itinerary. A mobile agent works autonomously and communicates with other agents and host systems. During the migration, the agent carries its code and some kind of execution state. On each host of the network, a MA platform is responsible for executing the mobile agent’s operations, providing a safe execution environment, and offering services for the MA. A MA system is the set of all MA platforms of the same type together with the MAs running on these platforms as part of an agent-based application. Many applications of mobile agent have been reported, including Electronic Commerce (Maes et al., 1999), Information Retrieval (Theilmann and Rothermel, 2000), Network Management (Bieszczad et al., 1998; Gschwind et al., 1999) and Mobile Computing (Takashio et al., 2001).

However, before we implement mobile agent-based applications, some important issues, such as fault tolerance, must be addressed. Many fault-tolerance schemes have been proposed for MAs, and one of the most popular solutions is the replication based scheme (Johansen et al., 1995; Shu et al., 2000; Komiya et al., 2002; Pears et al., 2003; Pleisch and Schiper, 2003). The basic idea of the replication based scheme is to maintain some replicas for the working MA. If the working MA fails and is detected by a replica, the replica will create a new working MA to continue the task. On the other hand, if the working MA detects that the replica failed, it will generate a new replica to replace the failed one. So the working MA and the replica will guard each other.

Existing replication based MA fault tolerance schemes have several shortcomings. The first is the overheads caused by the replicas. The replicas do nothing except monitor and synchronise with the working MA, which may increase the system’s overhead and slowdown the system execution dramatically. The second problem comes from the failure detection mechanism adopted in current replication schemes. The Heartbeat-Style Failure Detector (HBFD) is a well known failure detection technique. It requires the peers to keep on exchanging the heartbeat messages. This characteristic not only incurs message cost and causes false detection, but also needs modification for the MA environment, because no message can be delivered during a MA’s migration. We call this period the dumb period.

During the dumb period, the traditional heartbeats algorithm does not work properly and modification is needed. In this paper, we address these problems and propose new replication based algorithms for fault-tolerant mobile agent execution. Parallel processing is introduced in our proposed algorithms, which reduces the system’s overheads and improves the system’s performance. We also design the handover procedures to solve the dumb period problem. In order to reduce the message cost and the false detection caused by heartbeats, we propose a new approach to implement failure detector, which is named NTFD (Notification based FD). Instead of sending heartbeat messages periodically, NTFD sends failure notification messages only when a failure has been detected locally. Compared with HBFD, NTFD costs much fewer messages and guarantees the property of accuracy. To the best of our knowledge, this paper is the first work to study the implementation of failure detector for mobile agent fault tolerant execution.

The rest of the paper is organised as follows: Section 2 describes related works and the motivations for our research. Section 3 describes our proposed mobile agent fault tolerant execution algorithms. Two types of failure detectors are discussed in Section 4. Section 5 presents the analysis of the performance of the proposed algorithms, and validates the analysis result through simulation. The performance is compared with the well known rear-guard algorithm. Finally, Section 6 concludes this paper.

2 Related works and the motivations

Most of the replication-based MA execution algorithms in literature are based on the same rear-guard model. A working agent is followed by one or several replicas, called the rear guard agents (Johansen et al., 1995). If the working agent fails, the rear guard agent will continue the job for the failed agent. Later works made improvements on, and reported implementations of, this model. In Shu et al. (2000) presented a ‘sliding window’ mechanism. Before each migration of a MA, a specific number of backups of this MA are created in order to avoid the collapse or disappearance of this MA. In fact, the backups of the agent just play the role of rear guard agents. The size of the window is adjustable and determines the number of backups used. In Komiya et al. (2002), ‘surrogate of agent’ is used, which is just another name for a rear guard agent. A MA will leave a surrogate on each host that it visited. Once a surrogate finds out that the MA failed, it will recreate an
agent to continue the job. A mobile shadow scheme is
proposed in Pears et al. (2003), which employs a pair of
replica mobile agents, the master and the shadow. In Pleisch
and Schiper (2003) a pipelined model is proposed, in which
a witness agent follows behind a working agent. In fact,
both the shadow and the witness agent act as rear guard
agents.

The rear guard agent only guards the failure status of the
working agent, and keeps consistency with the working
agent in order to continue the work in case the working
agent fails. In order to improve the system performance,
we can let the replicated MA undertake tasks that can be
done concurrently with the working agent. In Cao et al.
(2003), the authors make the use of two reverse MAs to
eexecute in parallel by reverse itinerary to gain higher system
execution speed. In Qi et al. (2003), two MAs execute
reverse itinerary to speed up the execution and improve
fault tolerance. But these works focus on achieving load
balance and sensor networks’ performance respectively. The
fault tolerant execution of MA is not their main concern
which is, therefore, not addressed.

A problem common to existing works of replicated
MA execution is that they did not mention how to detect
failures. Failure detection (Chen et al., 2002; Nunes and
Jansch-Pôrto, 2004) is the mechanism necessary for
detecting the failure of an executing entity in the system.
Heartbeat-Style Failure Detectors (HBFD) have been widely
implemented in real systems. In Chen et al. (2002)
described how to configure a failure detector to satisfy the
required QoS. In Nunes and Jansch-Pôrto (2004) proposed
how to estimate the arrival time of heartbeat messages.
However, the conventional HBFDs have several problems
for mobile agent systems. In addition to dumb period and
high message cost, an unsolvable problem for HBFD is the
possible false detection due to the message delay or lost
messages. In Chen et al. (2002), authors studied the problem
of false detection in detail and proposed a set of quantitative
measures for its quality, including mistake recurrence time
and mistake duration. If a replica receives a false detection
from the failure detector, it may regenerate a new working
MA to replace the ‘failed’ one but, in fact, the working MA
has not failed. So it may cause duplicate execution.

In summary, although the rear-guard algorithm provides
fault tolerance for a MA system, it is not efficient. Also,
conventional HBFD are costly and the false detections will
cause extra trouble. All these problems will affect the
system performance. Many applications for data retrieval
applications, such as network management, need fast data
collection. Data submitted late usually are not useful, and
can even be harmful to the system. So fault-tolerance
algorithms should be efficient. We will describe our
proposed efficient algorithms for replication based MA fault
tolerance in Section 3.

3 Replication based MA FT algorithms

The main idea of improving the efficiency of replication
based mobile agent algorithms is to introduce parallel
processing among the replicas. According to whether the
MA’s itinerary is predefined or not, we propose two
algorithms, namely Reverse MAs Algorithm (RMAA) and
Alternate MAs Algorithm (AMAA).

3.1 RMAA

RMAA is well suited for MA applications with a predefined
itinerary and no requirement on the host visiting sequence.
One typical example is information retrieval. In RMAA,
the original predefined itinerary is the forward itinerary and
the reverse itinerary is an itinerary that reverses the
sequence of hosts in the forward itinerary. There are two
MAs in RMAA. One is called the Forward MA (FMA),
which will visit hosts according to the forward itinerary, and
another is called the Reverse MA (RMA), which will visit
hosts according to the reverse itinerary.

Figure 1 illustrates the RMAA scheme. The pair of MAs
is dispatched by the user’s mobile agent system at the same
time. They execute concurrently along their own itineraries
until they reach the two neighbouring hosts (e.g., Host 2 and
Host 3), which indicates that all the hosts on the itinerary
have been visited. The two MAs will then return to the MA
platform on the user’s host.

In order to prevent the failure of both MAs due to the failure
of a host, the two MAs are not allowed to land on the same
host. For this purpose, a landing procedure is needed
(Figure 2). The two MAs send the coordination message
‘Hello’ before migration to the next host. The ‘Hello’ message
is put into a queue on the MA platform of the next
host, which ensures that the host only accepts one MA with
the earlier ‘Hello’ in the pair MAs. A MA can migrate to the
host only if it has received an ‘Ok’ message as response
from the host. If both MAs send the ‘Hello’ message to the
host simultaneously, the host will receive both of them. But
in the queue, one will precede another. For the sender of the
later ‘Hello’ message, the MA platform will reply with a
‘No’ message. When the MA receives a ‘No’ message, it
knows that another MA is already on the neighbouring host.
So it will go back to the user host. The MA which got the
‘Ok’ message will return to the user host too after it finishes
its execution.

Similarly, with the rear-guard algorithm, we assume that
the FMA and the RMA will not fail at the same time.
During the execution of the pair of MAs in RMAA, one MA
may fail during its execution or migration. The failure detector will detect the failure and inform another MA, and the living MA will generate a new MA to replace the failed MA (Figure 3). For this purpose, FMA and RMA should keep each other’s computing results (this is the same with rear-guard algorithm). A distinguishing advantage of the RMAA algorithm is that it can handle the itinerary partition due to a link failure. In Figure 4, the itinerary is partitioned into two separated sections. It is obviously that the pair of MAs can finish their tasks if they will not fail at the same time.

**Algorithm 1** RMAA

```java
//RMAA is a class which implements all the functions of RMAA algorithm. User just needs to create a RMAA object and provide the Task and Itinerary to the RMAA object.
1. RMAA rmaa = new RMAA (Itinerary, Task); //RMAA creates 2 members: a FMA and a RMA;
2. rmaa.Launch(); //FMA and RMA are launched;
3. if (rmaa.FMA.tryMigration() = = OK) //will not encounter RMA
   rmaa.FMA.migration(); //migrate to next host
   result = rmaa.FMA.Task.start();
   rmaa.FMA.synchronize(result); //synchronize the computing result for failure handling.
   goto 3; //Finish the execution on current host, then try to go to next host.
} else //will encounter RMA if migrate to the next host. So FMA returns home.
   rmaa.FMA.returnHome();

// Pseudocode for the MA failure handling. Suppose ma gets a message from failure detector.
if (msg = ma.getMessage() = = MA_Failure) //get asynchronous message from failure detector
   ma1 = ma.clone(); //this ma will clone a new ma according to the failed ma’s information.
   ma1.migration(msg.host, failureMA_id); //the cloned ma migrates to the host.
   //After the cloned ma lands on the host, it will check the reported ma is really failed or not.
   if (ma1.check(failureMA_id) = = ReallyFailed) //if the reported ma really failed, its job will be
      ma1.resumeFailedma(); //continued.
```

RMAA can be implemented at the system level in a way that is transparent to the application programmer. What the programmer needs to do is just to provide the MA’s task and itinerary to RMAA. RMAA will create FMA and RMA to finish the users’ task. The algorithm in pseudo-code format for executing RMAA is illustrated in the Algorithm 1.

3.2 AMAA

A predefined itinerary is necessary for RMAA. But one of the fundamental features for the mobile agent is autonomy, which allows a MA to dynamically determine the next host to visit without a predefined itinerary. RMAA is not applicable in such a context while the rear-guard algorithm can still work. But the rear-guard algorithm is not efficient, so we seek a faster algorithm.

For a mobile agent application without a predefined itinerary, an agent needs to compute the next stop before every migration. Accordingly, we divide a MA’s operations into two sections (Figure 5): CalNextStopOps contains all the necessary operations that have to be done in order to
get the next stop; RestOps includes the rest of the operations (the de-registration operation at lest). The border between these two sections can be different for different applications. Some applications can determine the next stop in the first few steps; some get it at last.

**Figure 5** A MA’s operations

AMAA involves two MAs. One MA which is on the head is called Leading MA (LMA); the other MA which is behind the LMA is called Slave MA (SMA). The two MAs should arrange their operations in the two sections as described above. Figure 6 shows the execution process of AMAA. The MA platform on the user’s host launches two MAs. One lands on the first stop and becomes LMA. The other waiting on the user side becomes SMA. When the LMA has calculated the next stop, it sends a message to the SMA. SMA then migrates to the next stop and becomes the new LMA to start its execution. The former LMA becomes SMA now. When the LMA determines the next stop, it sends a message to SMA. Now the SMA may or may not finish the RestOps. When SMA finishes the RestOps, it will migrate to the next stop. The process will continue until the task is finished. As we can see, LMA and SMA execute at alternative hosts in the network.

**Figure 6** AMAA execution process (see online version for colours)

Similar to the rear-guard algorithm and RMAA, the failure detector will inform the failures of MAs, and the living MA will generate a new MA to replace the failed MA. However, different from RMAA, AMAA cannot handle the itinerary partition.

Same with RMAA, AMAA can also be implemented at the system level. Users need not provide the itinerary, but the task is required to separate into two sections as we described. Algorithm 2 illustrates the pseudo-code for AMAA.

AMAA can be easily extended to involve $n$ ($n \geq 2$) MAs. Among the $n$ MAs, One acts as the LMA. The rest $n - 1$ MAs form a sequence of SMAs. When the LMA determines the next stop, it informs the last SMA. The last SMA migrates to the next stop and becomes the LMA. The previous LMA becomes the first SMA in the sequence of SMAs.

**Algorithm 2** AMAA

```java
//AMAA is a class which implements all the functions of AMAA algorithm. User just needs to create //a AMAA object and provides the Task to the AMAA object.
1. AMAA ma[] = new AMAA (Task);//AMAA creates 2 members: an LMA and a SMA.
2. NextHost = FirstHost; ma[0].end = false; ma[1].end = false; //Initialization;
3. ma[0].goto 4; ma[1].goto 9;//ma[0] is current LMA and ma[1] is current SMA.
   //ma[0] and ma[1] share the same code from 4 to 9. In the following, “ma” can be ma[0] or ma[1];
4. ma.migration(NextHost); //When LMA determines the next stop, it informs the next SMA ma[1].
5. NextHost = ma.Task, CalNextStopOps();
6. if(NextHost != NULL)
   ma.informSMA(NextHost);//After the current SMA get
   else //to next host and becomes the new LMA. This
      //this message, it will migrate
   ma becomes the new LMA;
   (ma.informSMA(NULL); //No next host, so inform SMA
      //to return home.
   end = true; //This mark will make LMA return home
   1. result = ma.Task, RestOps(); //Finish the rest operations.
   2. ma.synchronize(result); //Synchronize the computing
   9. if( (end = true) //No next host.
      ma.returnHome();
      else if (ma.getNextHost() != NULL)//SMA get the next
   {goto 3; //host which is sent by LMA
      /SMA will migrate to the next host.
   } else {No next host
   ma.returnHome();//it is time to go home.
   } //Pseudocode for the MA failure handling in AMAA is the
      same with RMAA.
```

**4 Failure detection mechanisms**

The function of failure detection is a fundamental requirement for replication-based fault tolerance algorithms. HBFDs are widely implemented for realistic systems. But, as mentioned before, problems for HBFDs include the false detection, high message cost, and the dumb period for MA applications. For the problem of dumb period, we propose a handover procedure. For the false detection and message overheads of HBFD, we propose an alternative approach to implement failure detectors; called Notification Based Failure Detector (NTFD).

For the implementation of failure detectors, we consider the system consisting of a finite set of $N$ hosts ($N > 1$). On each host there are some processes which can be MA processes or system processes. A process can communicate with processes in the same host (intra-communication) by shared memory or messages. The intra-communication is reliable and synchronised. Hosts are connected by unreliable links. A process can communicate with processes on other hosts (inter-communication) by exchange messages. Messages can be delayed or lost. There is no global synchronisation mechanism for the system and each host’s
clock has no bound on the clock drift. However, we assume that the message delay and message losses follow some probabilistic distributions. For simplicity, we only consider that processes can fail by crash (prematurely halting). Hosts can be made reliable by the backup mechanisms. Failure detection is provided by a separate failure Detector Processes (FD). The FD collects the failure information of the MA processes being monitored on the same host and communicates with other FDs on other hosts.

4.1 Heartbeat-Style Failure Detectors (HBFD)

Figure 7 shows the system structure of HBFD. Each HBFD keeps collecting the status of the monitored MA in a predefined frequency on the same host. For each piece of collected status information, HBFD will construct a heartbeat message and send it to the interested FDs on other hosts recorded in the registration table.

Figure 7 HBFD

For example, in Figure 7, MA_B makes a registration on the HBFD_B of Host_B, while MA_A makes the registration on Host_A. The registration procedure has two steps:

- MA_A asks for the failure detection service from the local HBFD: HBFD_A. Within the service request description, MA_A nominates its replica MA’s ID and address (MA_B, Host_B).
- When HBFD_A received the registration request, it will allocate a new entry in the registration table for MA_A. The service requestor (MA_A) will be put in the first column of this row. The replica MA’s ID and address will be put in the following columns of the entry.

Before a MA migrates to a new host, it will make a deregistration on the current host. The deregistration procedure also has two steps:

- MA_A sends a deregistration request (stop monitoring the failure of MA_A) to HBFD_A
- HBFD_A will traverse the registration table to find the entry for MA_A ID and delete it (A complete registration table is shown in Figure 9).

In Figure 7, MA_B periodically updates its status on HBFD_B and HBFD_B sends these collected updates in the heartbeat messages to MA_A. If MA_B fails, the updating operation will stop. So there is no heartbeat message to HBFD_A and HBFD_A will assert a failure. The frequency of the updates and the heartbeat message can impact the QoS of the HBFD. For the implementation of a HBFD with certain degrees of QoS, please refer to Chen et al. (2002).

For the problem of dumb period, a handover procedure is needed. A simple solution is that before a MA starts migration, the ‘migration’ tag is piggybacked in a heartbeat message. When the failure detector monitoring the MA receives the message with the ‘migration’ tag, it will stop the failure detection for this MA and wait until it receives the new heartbeat message (at this time, the MA has landed on a new host). The problem for this scheme is that, if the MA is lost during migration, the failure detector cannot detect it. An enhanced scheme is based on reliable MA migration. When a MA starts a migration, it sends a replica to the next host and waits until the replica lands on the next host. During the migration process, the waiting MA can keep sending heartbeat messages to the failure detector. After the replica lands on the new host, it informs the waiting MA which will then hand over the task of heartbeat message exchanging to the replica. Through this scheme, the dumb period problem can be solved and the failure detector can keep working on the monitoring task.

False detection is an inherent problem of HBFDs. What we can do is to add a checking procedure. When a new MA is generated to replace the failed MA, the new MA should check the status of the failed MA. If the new MA finds a false detection, it will kill itself to avoid the duplicated execution. For the message overheads caused by HBFDs, we cannot reduce it only by decreasing the frequency of heartbeat messages exchanging, if we want to maintain quick failure detection. According to (Chen et al., 2002), decreasing the frequency of heartbeat message exchanging will make the time of failure detection longer. These two problems motivate us to design a new type of failure detector which can improve the accuracy property and reduce message overheads.

4.2 Notification-Style Failure Detectors (NTFD)

NTFD is an alternative approach to implementing the failure detector with 100% accuracy and very low message cost. For NTFD, instead of exchanging the heartbeat messages periodically, the underlying failure detection mechanism on a MA platform will detect the failure status of local MAs, and once a failure is detected, a failure notification will be sent to the replica. In this way, we shift the failure detection from the network domain to the local watchdog mechanism. Figure 8 shows the NTFD system. In a NTFD system, all the MAs running on a MA platform will be monitored by a NTFD running on the same platform.
Each NTFD maintains a registration table for the MAs that it is monitoring. Each MA will make a registration to record the address of its replica MA. Once a failure is detected, the NTFD will send a notification to the replica MA. For example, in Figure 8, MA_A and MA_B are each other’s replicas and they are monitored by NTFD_A and NTFD_B respectively. MA_A makes a registration in the registration table of NTFD_A and nominates MA_B as its replica MA. MAB does the same procedure on NTFD_B except that its replica MA is MA_A. Before a MA migrates to a new host, it will deregister at the current host. The registration procedure and the deregistration procedure are similar. (A complete registration table for NTFD is shown in Figure 9).

NTFD detects the failures of MAs through local failure detection mechanisms. The failure can be detected at machine instruction level, code module level, process level or system level. In this paper, as an example, we use the watchdog technique to detect local failures at process level (Figure 9). Watchdog is a variation of heartbeat style FD but it uses the shared memory to increase or decrease the HB (HeartBeat) counter, so no message passes between the monitored MA and the watchdog process.

The NTFD regularly decreases the HB counters in the registration table. The first column of the registration table is expanded to hold the HB counter. The monitored MAs are designed to reset their HB counters at particular points in their execution cycle. If the NTFD discovers that one of the HB counters has gone below a certain threshold during its decrementing operation, it signals an error and then sends the notification messages to all the replicated MAs for the failed MA.

The NTFD can be a software module and run on system level (kernel process) or user level (user process), and it can also be a hardware unit installed on the host. Watchdog is easy to construct. For example, the Linux kernel 2.4.x includes several watchdog drivers that include a software watchdog and drivers for several hardware watchdog boards. According to our model, all the processes are synchronised within a host. According to the discussion on the QoS of HBFD in Chen et al. (2002), there is no false detection of HBFD under a synchronised system.

NTFD also has the dumb period problem, so it needs the same handover procedure as in HBFD. But, compared with HBFD, NTFD is more efficient and scalable, and can guarantee the property of accuracy. However, a problem for NTFD is that it cannot guarantee the completeness property as well by HBFD, because the replica MA cannot acknowledge the failure if the failure notification message is lost. However, sometimes the accuracy property is more important. For example, in RMAA or AMAA, if one MA failed and the replica MA does not know the failure, the replica MA may finish the task by itself if it will not fail. But if HBFD makes a false detection, the replica MA may cause the duplicated operations, which is unacceptable in some applications like e-commerce. In Yang et al. (2006), we conclude that the overall performance of NTFD is better than HBFD. The algorithm, in pseudocode format for NTFD, is illustrated in Algorithm 3.

**Algorithm 3 NTFD**

```plaintext
// A NTFD gets a Request Msg for the failure detection service
if (msg = getMsg() == “failureStatusRequest”)
    if (watchdog = askWatchdog() == NULL)
        watchdog = OS.createWatchdog();
        watchdog.Add(msg.ProcessID);
    }

// An MA renews its HB counter
if (watchdog = askWatchdog() != NULL)
    watchdog.renewHBcounter(MA.ProcessID);
else
    PrintError();

// The Watchdog Process
while (proc = regTable.Process.hasNext(1))
    proc.HBcounter --;
    if (proc.HBcounter < 0)
        AssertFailure(proc.ID);

// Assert Failure for a failed MA: proc.ID
registeredProc = regTable.getRegisteredProc(proc.ID)
while (procID = registeredProc.nextProc())
    sendNotificationTo (ProcID);

// Registration table structure
struct regItem
    { int ProcID;
      int HBcounter;
      struct registeredProc * next;
      }

// struct registeredProc
struct registeredProc
    { int procID;
      struct registeredProc * next;
    }
```
5 Performance analyses and evaluations

In this section, we first make an analytic analysis of the execution time for the different fault tolerant MA execution algorithms, and then discuss the results of our simulation study.

5.1 Analysis on execution time and message cost

In the following discussion, we assume that the execution time $T$ for a MA is the same on each host. $N$ is the number of hosts. The time for a MA to migrate from the current host to the next stop is $T_m$. $T_{Task_exe}$ is the total execution time of each algorithm. For the rear-guard algorithm, the time taken by the migration, it will inform the rear guard MAs to follow it. We assume that the time needed for this operation is $T_{inform}$. In RMAA, $T_{landing}$ is the time needed by the landing procedure. In AMAA, like the rear-guard algorithm, $T_{inform}$ is the time needed by the operation of LMA informing SMA of the next stop, and according to Figure 5, we assume that the time taken by each CalNextStopOps is $T_{CalNextStopOps}$; the time taken by each RestOps is $T_{RestOps}$. It is obvious that $T = T_{CalNextStopOps} + T_{RestOps}$. For simplicity, we do not consider the cost of heartbeat messages and synchronisation messages, because they are needed by all of our discussed algorithms.

For the rear-guard algorithm, the whole task is finished by the single working MA and no parallel processing is involved (Figure 10). So we have: $T_{Task_exe} = N(T + T_m + T_{inform})$. For RMAA, the FMA and RMA execute in parallel, so ideally the execution time is:

$$T_{Task_exe} = N(T + T_m + T_{landing})/2.$$ AMAA allows partial parallelism in MA execution, and its execution time depends on how much of the job is done in parallel. From Figure 10, we can figure out how to compute the execution time for AMAA as shown in formulae (1) and (2). The parameter $T_{RestOps}$ determines the degree of parallelism that can be achieved. If we can increase the $T_{RestOps}$, the execution time of AMAA will be reduced. However, the reduction in the execution time is bounded by the condition that the total time will be no less than $(N + 1)(T_m + T + T_{inform})/2$, if $T_{RestOps}$ is greater than $T_m + T_{CalNextStopOps} + T_{inform}$. We define this $T_{RestOps}$ as the AMAA critical value. For AMAA involving $n$ MAs, it is easy to see that the task execution time will be

$$(N + 1)(T_m + T + T_{inform})/n, \quad 2 \leq n \leq N, \quad T_{RestOps} = T_m + T_{CalNextStopOps} + T_{inform}.$$ 

Table 1 summarises the execution modes, execution times and message costs for all the algorithms discussed in this paper. Note that for AMAA in Table 1, we assume that $T_{RestOps}$ is set as the critical value. We can see that RMAA can provide the fastest execution speed ($T_{inform}$ is almost the same with $T_{landing}$). But RMAA needs a predefined itinerary and also requires the system to allow a random hosts accessing sequence. These requirements make RMAA inflexible. AMAA has the same degree of flexibility as the rear-guard algorithm, but its execution time can only be shortened if the next stop can be calculated quickly (then $T_{RestOps}$ becomes bigger). If AMAA can only determine the next stop at the last step of its operation ($T_{RestOps} = 0$), the execution time will be the same as the rear-guard algorithm:

$$T_{Task_exe} = N(T_m + T_{CalNextStopOps} + T_{inform}) + T_{RestOps} = N(T_m + T + T_{inform}).$$ But normally the $T_{RestOps}$ will not be zero, because a MA has to perform some routing operations on a host at last, such as deregistration, release resources, etc. So we can always gain the partial parallelism so as to shorten the execution time. The results in Table 1 are just the theoretical values. In practice, the real execution time will be longer due to various overheads. We will compare this with the realistic execution time in the simulation.

### Table 1

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Itinerary</th>
<th>Execution mode</th>
<th>Theoretical execution time ($N$ hosts)</th>
<th>Message cost ($N$ hosts)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>2MAs</td>
<td>nMAs (n &gt; 2)</td>
</tr>
<tr>
<td>RearG MA</td>
<td>Self-initiate</td>
<td>Non-parallel</td>
<td>$N(T + T_m + T_{inform})$</td>
<td>$N(T + T_m + T_{inform})$</td>
</tr>
<tr>
<td>RMAA</td>
<td>Predefined</td>
<td>Full-parallel</td>
<td>$(N + 1)(T_m + T + T_{inform})/n$</td>
<td>$(N + 1)(T_m + T + T_{inform})/n$</td>
</tr>
<tr>
<td>AMAA</td>
<td>Self-initiate</td>
<td>Partial-paral</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The message cost of the algorithms mainly includes the coordination message cost for the synchronisation and for informing the next stop, which is illustrated in Table 1. Please note that the message cost in Table 1 does not consider the message cost caused by failure detector, because failure detection is a kind of service provided by the platform in our system. We will consider the message cost caused by failure detector in the simulation study.

5.2 Simulation results

In order to compare the realistic execution time in a real environment, we performed simulations of the Rear-Guard algorithm (RearG), RMAA and AMAA on the Naplet mobile agent platform (Refer to website: Naplet). For simplicity, we only implement two MAs in each algorithm. The simulations are carried out on a PC with Pentium 4 CPU (2.5 GHz), 256 MB RAM. The software environment is: Window XP, Java version 1.4, and Naplet mobile agent platform. Five Naplet mobile agent platforms are installed on the PC and we simulated the algorithms with the MA travelling 15, 25, 35, 45, 55, 65, 75, 85, 95, 105 nodes, respectively using different failure detection services (HBFD and NTFD). The number of MA failures is set to be 1/20 of the total number of hosts that have been visited and the failures are uniformly distributed along its itinerary, which indicates that for every 20 hosts, there is one failure. The exchange frequency of the heartbeat messages is five messages per second. For AMAA, we set the $T_{\text{HistOps}}$ to be its critical value, which means that a MA will send out the next stop message in the middle of its execution.

From the simulation results in Figures 11 and 12, we can see that the execution time of RMAA and AMAA takes about half of the rear-guard algorithm’s execution time. But AMAA is a little slower than RMAA due to the partial parallelism and the synchronisation cost, such as informing of the next stop. Comparing Figures 11 and 12 we can see that using different failure detectors will not make much difference in the algorithms’ execution time. This is because the interface between a monitored MA and HBFD is almost the same as the interface between a monitored MA and NTFD in our implementation, which results in the similar execution time for an algorithm using different failure detector.

However, different types of failure detectors make big differences on the message cost. Figure 13 shows that the message cost increases much more quickly with HBFD. This is because the heartbeat messages’ cost is in direct proportion to the execution time and takes up the most part of the exchanged messages during a MA’s execution. Longer execution time will cause more heartbeat messages. But the message cost caused by NTFD becomes very low, because the NTFD’s message cost has no relation with the execution time but is in proportion to the times of MA failure. Under NTFD, the message cost mostly depends on the fault tolerant algorithms’ own message cost (Table 1). That is why, in Figure 13, the message costs of Rear guard MA and AMAA nearly overlap, because their own message costs are all $2N$.

6 Conclusions

In this paper, we first proposed two efficient replication-based execution algorithms for fault tolerant MA execution. The algorithms allow for parallel processing and provide tolerance of MA failure. Then we discussed the failure detection techniques for replication-based MA fault
tolerance algorithms. Based on the shortcomings identified for the traditional HBFD, we proposed a new type of failure detector: NTFD. Analytic and simulation results show that the proposed algorithms can improve the system’s execution speed dramatically. The shorter execution time can help MA to bypass host failures with greater probability and reduce the number of heartbeat messages exchanged. NTFD can help us reduce the message overhead to a very low level and, thus, is very useful for building a high-performance fault tolerant MA system.

Acknowledgements

This work is supported in part by the University Grant Council of Hong Kong under the CERG Grant PolyU 5075/02E and China National 973 Program Grant 2002CB312002.

References


Website


Appendix Implementation in a real system

/*****************************/
A General MA(G-MA) class. It has the basic functions for a MA

/*************************************/
public abstract class GeneralMA
{
  public GeneralMA (Itinerary iti, Tasks t) {...}
  public void MigrateToNext (Addr addr) {...}
  public void LandOn () {...}
  public void ProcessingMsg (Message msg) {...}
}

Implementation of RMAA

RMAA is independent of MA platform. RMAA is a class which implements all the functions of RMAA algorithm. The user just needs to create a RMAA object and provide the Task and Itinerary to the RMAA object.

/*****************************/
RMAA_MA class

/*************************************/
public class RMAA_MA extends GeneralMA {
    RMAA_MA peerMA;
    State state;
    public RMAA_MA(Itinerary iti, Tasks t) {
        super(iti, t);
    }
    public void AssignPeerMA(RMAA_MA peer) {
        peerMA = peer;
    }
    public void MigrateToNext(Boolean fixfailure, Addr addr) {
        if (fixfailure) {
            super.MigrateToNext(addr); //call G-MA’s migration process
            state = fixfailure;
        } else {
            if (CkMeeting() != Yes) //check the meeting event
                result = sync(peerMA.ID); //synchronize the computing results
            super.MigrateToNext(); //call G-MA’s migration process
        } else
            BackHome(); //if the MAs meet, both MAs return home.
    }
    public void LandOn () {
        //After the cloned ma lands on the host, it will check the reported ma is really failed or not.
        if (state == fixfailure AND Check(peerMA) == Really Failed) {
            flush(); //if the reported ma really failed, its job will be continued
            resume();
        } else
            super.LandOn();
    }
    public void ProcessingMsg (Message msg) {
        if (msg == MA_Failure) //get asynchronous message from failure detector
            ma = clone(result); //this ma will clone a new ma according to the failed ma’s information.
            ma. MigrateToNext (true, msg.addr); //the cloned ma migrates to the host.
    }
}

/* *****************************************************************/
RMAA class
/* *****************************************************************/
public class RMAA {
    RMAA_MA fma, rma;
    public RMAA(Itinerary iti, Tasks t) {
        iti = getForwardIti();
        fma = new RMAA_MA(iti, t);
        iti = getReverseIti();
        rma = new RMAA_MA(iti, t);
        fma.AssignPeerMA(rma);
        rma.AssignPeerMA(fma);
    }
    public Launch() {
        fma. MigrateToNext ();
        rma. MigrateToNext ();
    }
}
/* *****************************************************************/
Using RMAA class in user applications
/* *****************************************************************/
1. RMAA rmaa = new RMAA (Itinerary, Task); //RMAA creates two members: a FMA and a RMA;
2. rmaa.Launch(); //FMA and RMA are launched;
Efficient algorithms for fault tolerant mobile agent execution

Implementation of AMAA

AMAA is independent of MA platform. AMAA is a class which implements all the functions of AMAA algorithms. The user just needs to create an AMAA object and provide the Task and Itinerary to the AMAA object.

/********************************************
AMAA_MA class
********************************************/

```java
public class AMAA_MA extends GeneralMA
{
    AMAA_MA peerMA;
    State state;
    AMAA_MA(iti, Tasks t)
    {
        super(iti, t);
    }
    public void AssignPeerMA(AMAA_MA, peer)
    {
        peerMA = peer;
    }
    public void MigrateToNext(Boolean fixfailure, Addr addr)
    {
        if (fixfailure == true)
        {
            super.MigrateToNext(addr); //call G-MA's migration process
            state = fixfailure;
        }
        else
        {
            if (CkMeeting() != Yes) //check the meeting event
            {
                result = sync(peerMA.ID); //synchronize the computing results
                super.MigrateToNext(); //call GeneralMA's migration process
            }
            else
            BackHome(); //if the MAs meet, both MAs return home.
        }
    }
    public void LandOn()
    {
        //After the cloned ma lands on the host, it will check the reported ma is really failed or not.
        if (state = fixfailure AND Check(peerMA) = ReallyFailed)
        {flush(); //if the reported ma really failed, its job will be continued
            resume();
        }
        else
        super.LandOn();
    }
    public void ProcessingMsg (Message msg)
    {
        if (msg = MA_Failure) //get asynchronous message from failure detector
        {ma = clone(result); //this ma will clone a new ma according to the failed ma's information.
            ma. MigrateToNext (true, msg.addr); //the cloned ma migrates to the host.
        }
    }
    public void NotifyNextStop()
    {
        msg = new Message(NextStop, peerMA);
        SendMsg(msg);
    }
    public void waitNotification(Message msg)
    {
        NextStopAddr = msg.addr;
    }
}
```

```
public class AMAA
{
    AMAA_MA lma, sma;
    AMAA(Addr First_host_addr, Tasks t)
    {
        super.LandOn();
    }
```

lma = new AMAA_MA(First_host_addr, t);
sma = new AMAA_MA(null, t);
lma.AssignPeerMA(sma);
sma.AssignPeerMA(lma);
}

public Launch()
{
    lma. MigrateToNext ();
    rma. waitNotification();
}

/********************************************
Using AMAA class in user applications
********************************************/

1. AMAA amaa = new AMAA (NULL, Task); //AMAA creates two members: a FMA and a RMA;
2. amaa.Launch(); //LMA and SMA are launched;