On Handling Component and Transaction Failures in Multi Agent Systems

Pradeep Reddy Varakantham Santosh Kumar Gangwani Kamalakar Karlapalem
International Institute of Information Technology, Gachibowli, Hyderabad, INDIA
pradeep@gdit.iiit.net santosh_g@gdit.iiit.net kamal@iiit.net

Multi agent systems are being used for various practical applications like e-commerce, e-auctions and gathering information from the web. Thus there is a need for these systems to be robust. However, agents can fail due to component failures. The atomic tasks taken up by the agents might also fail. So, agents need to recover to a correct state after a failure. This paper deals with the logging required and hence forth the recovery protocol to recover the agent to a correct state. An agent is modeled as a colored Petrinet. By logging the state information of the agent (Petrinet) an agent recovers from failures. For atomicity of tasks in multi-agent systems, linear two-phase commit protocol with some enhancements is proposed

Keywords
Agent, multi agent system, recovery in agents, logging in agents, atomicity in agents, durability in agents.

1. INTRODUCTION

Multi agent systems are being used for various practical applications like e-commerce, e-auctions and gathering information from the web. Given the critical nature of these applications these systems must be robust. However, agents can fail due to failures of components, such as agents, communication channels or computers referred to as component failures or the atomic tasks taken up by the agents might fail (failures caused by the transaction) referred to as transaction failures as mentioned in [1]. So, agents need to be able to recover to a correct state after failure. In this paper we deal with recovery in case of both transaction and component failures.

We consider ACID (Atomicity, Consistency, Isolation, Durability) like properties [2] for agent transactions. Existing work [1] deals with isolation. It deals with the dependency between sub-tasks of a task in an agent because one sub-task accessed the data or results of another sub-task violating the isolation property. Based on this dependency, it determines if a sub-task fails what other sub-tasks should be aborted. It assumes that each agent in the transaction hierarchy must log the status of all it's ancestors and descendents. This is against the distributed nature of multi-agent systems and the dynamic hierarchical nature of their tasks. [1] also suggests 2 phase commit (2PC) protocol for atomicity in agents, but does not give any details about how it is suitable to agents and if any changes are required in 2PC for agents. In agent systems to the best of our knowledge no other work deals with logging and atomicity. Consistency is an important issue in agents but difficult to handle because of the requirement of domain specific knowledge. In [13] beliefs are copied to a separate agent (sentinel) from the agent system for early detection of faulty agent and of inconsistency between agents. [7] deals with failure handling for transaction hierarchies. [12] deals with exception handling in agents.

1.1 Our Contributions

This paper deals with recovery in a multi-agent system. Atomicity of a task in an agent means that the task is done in it's entirety or not done at all. Durability of a task means that the changes of the task after the task is completed must persist and if an agent takes up a commitment it will not abandon it because of a failure. The recovery algorithm ensures atomicity and durability of tasks in the agent when failure occurs. Isolation of a task means that the task should be executed as though it is being executed in isolation from other concurrent tasks. For isolation the agent designer has to specify when the resources required by the task are reserved. The granularity of reservation (locking) and the modes of reservation (like read and write locks) are not dealt here. The release of the resources is done at the end of the task. We do not deal with consistency of tasks. We deal with the logging required for recovery of agents in case of failures. These log records help in recovery from both component failures and transaction failures. Our state logging is similar to storing process state before process switch. Our recovery in agents is similar to shadow page recovery, see [8].

[9] and [10] use Petrinet for the definition of a very abstract form of multi-agent systems. In [4], using an object-oriented approach and colored Petrinets (see [3]) a generic agent class is modeled. [4] considers the tasks of the agent as methods and treats their execution as atomic. In this paper, the assumption is made that each task consists of atomic actions whose sequence and execution can be modeled as part of the colored Petrinet representation of the agent.

We design a distributed logging scheme for the state information of the agent to recover from failures. It is suitable to multi-agent systems because multi-agent systems have autonomous agents, have hierarchical delegation of tasks, and are mostly distributed. The disadvantage is that each agent must take care of logging and recovery.

Recovery can also be done in a multi-agent system by logging in one or more separate (recovery) agents the state of the entire multi-agent system. The advantage being that all the agents in the multi-agent system need not handle logging and recovery. But there should be an interface to the recovery agent(s) for logging information in the recovery agent(s). The disadvantages are that it goes against the autonomy of the agent, applies a centralized solution, and security and privacy of information may get compromised.

To ensure atomicity of tasks in a multi-agent system a modified linear two phase commit (2PC) [10] is proposed. This is suitable for multi-agent systems because of the autonomous nature of the agents and the dynamic hierarchical delegation of tasks in multi-agent systems.

2. Assumptions and Terminology

We concentrate on the basic features of an agent which are given below:

Beliefs: What an agent believes about itself and about it's environment?

Tasks: What work an agent performs?

Capabilities: What tasks an agent is capable of doing under current conditions?

Commitments: What are an agent's obligations towards the environment and to itself?

As our focus is on recovery, we will abstract away from the many details of the agent model. For agent models, see Shoham's work [6] and BDI agents [5]. A task is a basic unit of work performed by an agent. It consists of a set of atomic actions. A task is like the set of actions in the behavior rule in Shoham's model [6]. An action can be either a *private* action or a *communicative* action. A private action is one that interacts with the environment. A communicative action is sending or receiving a message. An assumption is made that an agent receives commitments and if it has the capability to do the requested task, it tries to meet the commitments received. We assume that requests for both self-commitments and commitments from other agents are made through messages. A **message** object that is used for communication between agents is composed of the following fields:

Field	**Description**
Sending agent(sa)	The ID of the agent that sends the message
Receiver agent(ra)	The ID of the agent that should receive the message
Message ID(MID)	The ID of the message that is being sent. It should be unique for an agent. It can be made up of the task and the action in the task which sent the message in the task.
Response ID (RID)	If the message is a response to an earlier message sent by this agent then the response ID is the message ID of the original message sent by this task. The response ID is zero if it is a request for a task.
Action (ACT)	The task that the receiver agent should perform.
Content (content)	Indicates the content of the message.

3. AGENT MODELING USING PETRINET

Agents can be modeled using various formalisms such as finite state automaton. But since many tasks are performed by an agent at the same time we need to model parallel processes. Therefore, an agent is modeled by using Petrinets. In 3.1, we present a generic Petrinet model of an agent and in 3.2 we provide an example for modeling an agent as a Petrinet.

3.1 Generic Petrinet model

The Generic Model of an agent as a Petrinet is shown in Figure 1. An agent is initially in a state waiting for some message (P0). Whenever it receives a message (it can be from the same agent or from other agents) it goes to P1 state through transition T10. It then checks the response ID. If this response ID is not set to zero then it is a response message and so transition T2 is activated. The response ID is made up of the task and the action (place) which sent the original message.

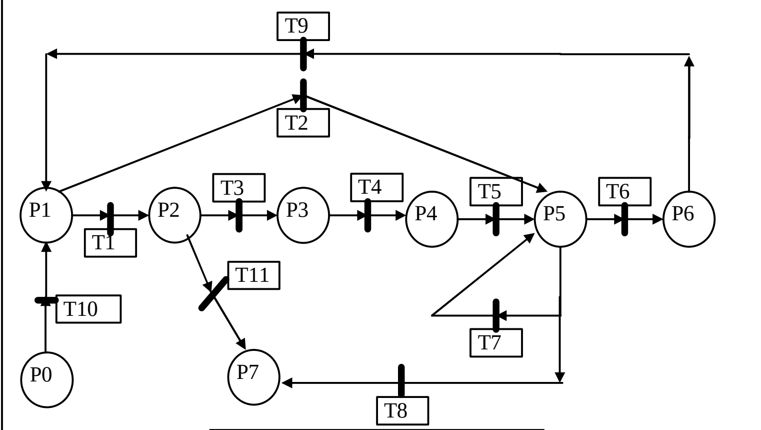


Figure 1: Agent modeling using Petrinet

P0 – Waiting state, T10 – receiving message, P1 – received message, T2 – checking if this is a response to an earlier message and if it is forwarding to the task that sent the message, T1 – if the message is a request for a task, P2 – the message is a request for a task, T3 – check whether the agent has the capability for doing the task, P3 – agent has the capability to do the task, T11 – the agent does not have the capability to do the task, P7 – end place, T4 – agent sends commitment accepted message, P4 – Commitment accepted, T5 – starting the task, P5 – the task sub-Petrinet, T6 – want to send a message, P6 – message handling, T9 – sending message.

The agent moves into the place (in the part of the Petrinet represented by P5) of the task that sent the initial message to which this is a response through transition T2. If the response ID is zero it means that this is a message for performing a task and the agent moves into the place P2. In the place P2 the agent checks whether it has the capability to perform the *ACT* that it has received in the message. Checking the capability involves checking if the beliefs of the agent allow it to take up the task or not. The beliefs that may need to be checked include beliefs about the tasks the agent can take up (knowledge about itself) and the beliefs about the requesting agent (the requesting agent’s permissions for security purposes). If it does not have the capability it sends a negative response to the sending agent and goes to the end place, P7. If it has the capability, it moves to the place P3. It then gives the acceptance of the commitment to the agent that sent the message and moves to the place P4. From P4 it moves to the part of the Petrinet consisting of actions necessary to perform the task for this commitment. Here we have taken this as a single place P5 for simplicity. It may contain any complex set of actions represented by any sub-Petrinet instead of the place P5. The Petrinet should contain actions for reserving the resources required by the task. There is a variable associated with each P5 place namely **ActionNum** indicating the action each is about to execute in the sub-task if more than one action can be performed in that place. This variable is incremented whenever the agent enters this state. Its initial value is set to 0. When the agent has to send a message it fills in the particulars of the message object, sends it and moves to the place, P6. In P6, the agent transmits the message to the receiver using a 2-way handshake. After the task is complete it goes to the end place P7.

3.2 Example: Search agent

In any agent only the part of the Petrinet corresponding to the place P5 changes. So, only this part of the agent along with the place P4 is shown in Figure 2 for a search agent. Its work is to search for a book in a digital library. So, it has a belief saying it is capable of taking up ‘search’ tasks. It also has a belief stating it can accept search tasks from all agents. It has the task of searching the digital library, which consists of say two databases.

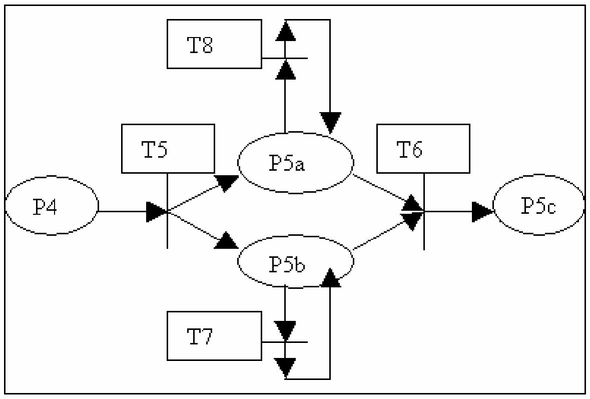


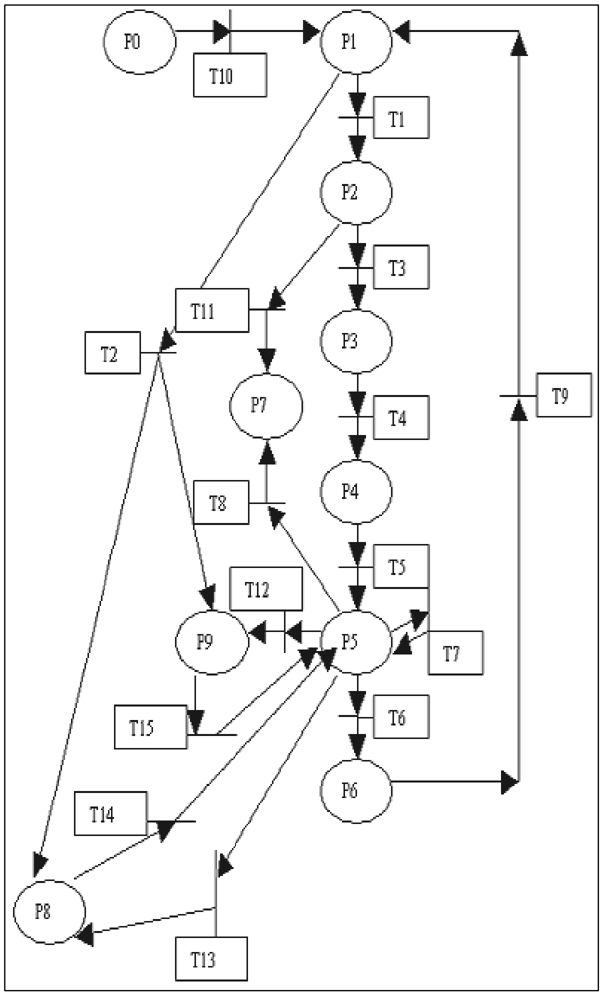
Figure 2 Search Agent

Thus a search of the library will be a search in the first database concurrently executing with the search in the second database. The results are combined in the place P5c. The result is sent to the agent which made the request through P6. The search task consists of the following actions for both P5a and P5b: (i) Connecting to the database to be searched, and (ii) Perform the search on the database, and return the results. The two actions are assumed to be atomic. Both P5a and P5b have **ActionNum** variable which is initially zero. **T6** is the synchronization transition where the results are synchronized. **T7** and **T8** indicate the execution of actions. During these transitions the **ActionNum** variables in the P5a and P5b places are incremented.

3.3 Example: Buy Agent

A buy agent has the task of performing the steps required when a customer wants to buy a book. So, it has a belief saying it is capable of taking up ‘buy’ tasks. It also has a belief stating it can accept buy tasks from all other agents. The information that the customer gives to the agent includes the book name, the delivery address and the credit card number. This agent checks if it has the capability to perform the ‘buy’ task. It then searches the book database to obtain the information regarding the price of the book and the number of copies. The work of the searching is done by the S**earch Agent**. If the available number of copies are more than the number of copies required the buy agent reserves the books. If the books are not available, it will check if the required number of books are available but are reserved for other agents. If it is so, it will wait in a queue till the transaction which has reserved the books completes or a timeout occurs. If that transaction fails the first transaction in the queue can reserve these books. If books cannot be reserved an error is sent to the requesting agent. After reserving the books the credit card is validated for the buying process to continue. If the credit card is validated then the books are dispatched by the **Dispatch Agent** to the customer's address. The copies dispatched are removed from the books database. The Petrinet for the Buy agent is given in Figure 3. The Petrinet that is to be substituted at the state P5 in the agent Petrinet has places P5, P8 and P9 and transitions T12, T13, T14 and T15. The places P8 and P9 are places used for waiting after sending the message. This agent sends the book name to the search agent to check if it is present and moves to place P8. P8 is the waiting state used to receive the response of the search agent. The response of the search agent will have the response ID as the ID of this task (MID) and the action (P8), so the agent after receiving this response message forwards it to place P8 using transition T2. This also happens at the place P9. If the response is positive, that is, if the book is present in the database and if resources are reserved successfully T14 is activated and after that the agent moves to place P5, else a negative acknowledgement is sent to the sending agent that the task was not successful (this is not shown in the Petrinet due to space constraints). Similarly P9 is the waiting state for the dispatch agent. If the dispatching goes through smoothly then the response is positive and hence the agent moves into the P5 state. Other actions are performed by being in the P5 state itself. The important action that is performed in the P5 state is the credit card validation. Credit card validation includes checking if the credit card number is valid and checking if there is sufficient money to satisfy the order placed.

Figure 3 Buy Agent



4. SINGLE AGENT RECOVERY

We present the type of failures, model state of an agent, develop a logging scheme, and then give the recovery protocol for single agent failure.

4.1. Types of Failures

We deal with two types of failures in an agent as described in [1]:

Component failure: The agent becomes unavailable for some time due to the failure of a component, such as an agent, or a computer, and loses its volatile memory contents. Durability of a task means that the changes of the task after the task is completed must persist and if an agent takes up a commitment it will not abandon it because of a failure. When an agent makes a commitment to itself or any other agent it should perform the task required to make the commitment true. If an agent fails, on recovery it should still do the tasks corresponding to commitments made before failure if the commitments have not been withdrawn. The redo of actions after recovery should be minimized.

Example: An agent is asked to look up a list of books and it takes up the commitment to do it. Now if a failure occurs, the agent should find the list of books after it recovers if the commitment has not been withdrawn. This task should be performed with minimum possible redo of actions. (If half the list of books was looked up before failure, looking over it again should be avoided after recovery to the extent possible).

If a commitment is completed it's changes should not be lost. If a commitment is withdrawn all the actions already performed for this commitment should be withdrawn. **Example:** If an agent is asked to add money to an account and it takes up this commitment. If the commitment completes the change should be visible in the account even if failures occur. If the commitment is withdrawn the agent should be able to undo actions already done for this commitment.

Transaction Failure: Suppose an action fails in the set of actions corresponding to the task being performed for a commitment. Atomicity of a task in an agent means that the task is done in it's entirety or not done at all. In an agent a task may be said to have completed in it's entirety even if some non-critical action in it fails. So, only if the failed action is critical for the task, then the agent may have to undo all other actions already done and try the same task again or try some other task for the commitment or give up the commitment altogether.

Example: For a bank agent if the task is to transfer money from one account to another, actions are (i) Deduct amount from account 1 and (ii) Add amount to account 2.

Here atomicity of the task is a must. So, if any one of the above actions fail then the agent has to undo the whole task. If the failed action is not critical for the task, then the task may simply continue.

Example: Look up for a book in different databases, the actions for this task are (i) Look up in company 1's databases, and (ii) Look up in company 2's databases. Now if company 1's databases are not available, it can continue and look in the other database. Here atomicity is not required. The actions may still have to be undone if the commitment is withdrawn. In recovery, we should

- Ensure that the commitments completed are not lost because of failures (Durability) with minimum redo of work.
- Undo actions already done for commitments withdrawn (Durability).
- Perform tasks corresponding to commitments made before failure (Durability).
- Undo actions already done if a critical action fails in an atomic task (Atomicity).

4.2 State of an agent

Execution state of an agent constitutes the following: (i) Commitments made by the agent and still active (MIDs) (ii) Current active places in the Petrinet for each commitment (iii) Variables with values in the Petrinet (Ex: **ActionNum** in the P5 place) for each commitment and (iv) Data for each commitment consisting of variables, changed database parts (like tuples and tables), documents, information on resources reserved, etc. The Petrinet here serves as providing a formal state to the agent at every instant in it's execution. For performing tasks corresponding to commitments made before failure, the agent needs to log the commitments in a commitments log before they are made. For minimum redo of work, the agent needs to log the progress of commitments at regular intervals of time (snapshot of the commitments), which we can call the checkpoint or the snapshot point. The agent writes the snapshot (state of the agent) in the 'Agent State log' (See Figure 4). So every agent has a commitments log and an agent state log. Since actions can be complex and compensating actions are not given, undo may not be possible. So, for making undo possible, changes made by a commitment are not reflected in their place till the commitment is complete. The agent instead writes the state of each commitment in the log at check point. When a commitment ends or an atomic task is done completely and no withdrawal can take place, the agent is considered to have satisfied its commitment. The changes made by the commitment are reflected in their place now. The resources reserved by the commitment are released.

4.3 Logging in an Agent

We assume the messages are present in a message pool. Each message has a MID (message ID). The message ID is unique for an agent. It is made up of the task and the action (place in the Petrinet) which sent the message in the task. To make it unique in the multi-agent system, we append to it the sending agent's ID.
The procedure for logging is as follows

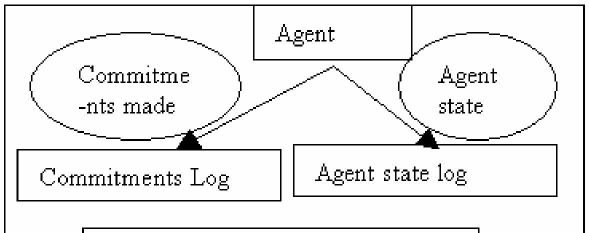


Figure 4 : Logging in an agent

1. Writing Commitments: As soon as we get a message from the message pool we check if a message with the same MID (Message ID) has arrived before. If the message has arrived before just drop the message and see the next message. If the message has not arrived before, continue with the processing of the message. The message contents are kept in the data for this message along with it's ID, so that it can be accessed later if required. If the agent has the capability for the requested task it writes 'COMMITMENT MADE' in the 'commitments log' and a 'commitment made' message is sent to the requesting agent.

2. Writing Agent state: At regular intervals of time we have the checkpoints. At checkpoints, we log the state of the agent in the 'agent state log' as follows. For the message IDs for which commitment has been made and not yet completed (neither successfully completed nor aborted), we write the message id and the state of the commitment in 'agent state log'. Log Record for each commitment will be as follows:

<MID, {places active}, {(variable of Petrinet, value of variable)}, link to the data for this task>

The {} indicate zero or more values.

Data consists of variables, database parts, documents, etc. The algorithm for automatic log record generation is given in **Table 1**.

Search Agent Example: For figure 2, search agent, suppose there is a message to search a book named “Agents” with MID 1. The agent find it’s Response ID is 0. It checks it’s beliefs and finds it has the capability of performing the ‘search’ task. It also checks if the requesting agent has the permission for the ‘search’ task.

Table 1: Automatic Log Records Generation

```
Procedure (Automatic LOG RECORDS Generation)
Input : Agent Petrinet(AP), events occurring during agent execution
Output : Log records generated
Procedure:
begin
Construct generic Petrinet;  /* As explained in section 3 */
 /* Substituting the agent Petrinet at state P5 */
 Substitute (agent Petrinet,P5);
 /* while agent is executing */
 WHILE(!EndExecution(agent))
 begin
  /* when an event occurs in the execution of an agent :
we need the place reached due to the event and the transition by which the place was reached */
  (place, transition) = Event_Occurred(AP);
   SWITCH(place,event) :
   begin
    case ‘P5’,’T2’ : /* P5 is the place and T2 is the transition*/
      Write(“MID End”,CommitmentsLog);
       break;
    case ‘P7’,’T11’ :
       Write(“MID End”,CommitmentsLog);
       break;
    case ‘P4’,’T4’ :
      Write(“MID Commitment Made”,CommitmentsLog);
      break;
    case ‘P7’,’T8’ :
      Write(“MID End”,CommitmentsLog);
       Break;
     /*  “*” indicates for any place */
    case ‘*’, ‘Withdraw or failure of action for an atomic task’ :
      Write(“MID Undo”,CommitmentsLog);
       break;
    case ‘*’, ‘checkpoint’ :
       takeSnapshot();
      break;
   end
 end
end

Procedure takeSnapshot()
begin
/* get space for new agent state */
/*state consititutes places active, variables etc.*/
  newState = GetNewAgentState();
  prevState = GetStateFromAgentStateLog();
  Write(“agent state start”,CommitmentsLog);
  Write(link(newState),CommitmentsLog);
/* getting the commitments for which tasks are partially done */
activeCommitments = GetActiveCommitments();
/* writing the state of all active commitments */
For each commitment MID in the activeCommitments
Begin
 Write(<MID {placesActive} {(variableName,    variableValue)} taskDataLink>,AgentStateLog);
end
/* writing all commitments completed */
 For each commitment,MID for which MID END present in  commitments log
 begin
 /* if the commitment has made any changes in the internal  state of the agent and / or the agent's environment */
```

```
  If (*IsEffectedEnvironmentorInternalState*(MID))
  *begin*
   *Write*(<MID {placesActive} {(variableName, variableValue)} taskDataLink>,AgentStateLog);
    /* Reflect in place the changes made by the commitments to the internal state of the agent and/or the external environment using log */
   *ReflectChangesInPlace*();
  *end*
 *end*
*Write*("agent state end",CommitmentsLog);
 /* Delete the previous state of the agent from the agent state  log */
 *Delete*(prevState,AgentStateLog);
 /* Delete the portion in 'commitments log' before agent state start because we have the information of all commitments before this in the 'agent states' log */
 prevData=*DataBeforeAgentStateStart*(CommitmentsLog);
 *Delete*(prevData,CommitmentsLog);
 /* delete the messages in the message box for which state is recorded in the current state */
 *DeleteMessages*(MessageBox, activeCommitments);
 /* delete messages for tasks undone and completed*/
 *DeleteMessagesUndoneCompleted*();
*End*
```

Since there is a belief which says any agent can request this task the agent takes up the task. It writes into the commitments log the record, <1 commitmentmade> and sends ‘commitment accepted’ to the requesting agent. It reserves any resources required for the task like the books table in the database. It starts the task of searching. There is another message for searching a book named “recovery” with MID 2. It finds the RID zero and finds that the agent has the capability for the searching task as in the previous case. It writes into the commitments log the record <2 commitmentmade> and sends acceptance of the commitment. There is a third message for searching a book named “multi-agent system” with MID 3. The agent finds the RID zero and finds that the agent has the capability for the commitment and writes into the commitments log the record <3 commitmentmade> and sends acceptance of the commitment. The task corresponding to MID 1 has ended. So, a log record <1 end> is written in the ‘commitments log’. At checkpoint, the active places are P5a and P5b for both MID 2 and MID 3. At checkpoint, in commitments log, the log record <agent state start> is written. In ‘agent state log’, it writes the MID, places active, variables and their values and link to data for the commitment which includes what resources if any were reserved. For MID 2, connecting to both the databases is done but querying has not started. For MID 3, connecting to the first database is done and connecting to the second database is not done but querying has not started for both the databases. So, in ‘agent state log’ the log records are

<2, {P5a, P5b}, {(ActionNumA, 2), (ActionNumB, 2)}, DataLink>, <3, {P5a, P5b}, {(ActionNumA, 2), (ActionNumB, 1)}, DataLink>

For MID 3, the value 2 for ActionNumA indicates that first action is done in the place P5a and the value 1 for ActionNumB indicates that no action has been done in the place P5b. For MID 3, data will contain a handler to the database connected by the first action in P5a place. For MID 2, for both P5a and P5b the first action has been done and Data will contain two database handlers. At checkpoint, the task corresponding to MID 1 is complete. Since this commitment does not change anything in the agent’s environment or it’s internal state (mental state) this commitment is not logged in the ‘agent states log’. Now say a new search commitment with MID 4 is made. The agent writes a log record <4 commitmentmade> in the ‘commitments log’. After the state is recorded it writes a log record <agent state end> in the ‘commitments log’. It discards the log before <agent state start> after making the changes of the completed commitments in the agent’s mental state and it’s environment. It discards the previous checkpoint. It drops the messages corresponding to MID 1, MID 2 and MID 3 from the message pool since 1 is complete and for 2 and 3 it has recorded the state. Now the following records are in the ‘commitments log’, “<agent state start>, <4 commitmentmade>, <agent state end>”. Now a message with MID 5 arrives. Now suppose failure occurs in the search agent, the recovery for this failure is presented in the next section.

4.3.1 Issues in handling Idempotence

Idempotence of an operation implies that the execution of the operation over and over is equivalent to executing it once. An agent’s actions are complex and can be stated at a higher level, so they need not be idempotent. Therefore, just redoing a set of actions cannot do agent recovery because the actions may not be repeatable. After commit, the human actions like dispatching a book can be done by humans (treated as another agent). But actions involving storage have to be reflected in their place using the log, that is, database parts should be updated, documents should be updated, beliefs should be updated, etc. This involves overwriting the initial stable storage parts by the changed data present in the data of the task in the log. This overwriting of the data can be repeated (the changed data can be written to the same addresses from the same addresses of the stable storage as many times as required). So, this writing to the stable storage is idempotent even though the actions themselves may not be idempotent.

Table 2 : Recovery Algorithm

```
Algorithm (Recovery):
Input: Commitments Log, Agent States Log and Message Pool.
Output : Recover the agent from failure
Procedure:
begin
/* Indicates tracing the commitments log
starting from the end */
 TraceCommitmentsLog();
 Recover();
end
/* getting the commitments active, commitments completed, commitments undone and the latest agent state link from the
commitments log */
Procedure TraceCommitmentsLog()
begin
 /* initializing counters of commitments active, completed and undone. */
 activecounter = 0;
 completecounter = 0;
 undonecounter = 0;
 /* complete agent state not yet seen */
 AgentStateComplete = false;
/* starting from the end of the commitments log*/
CommitmentLog (seek(GoToEnd)) ;

/* from last record to the record 'agent state start'. This should have a corresponding 'agent state end' before for the state to
be complete */
From LastRecord
until record = "Agent State Start" AND     AgentStateComplete = true
begin
/* if last agent state was recorded completely */
 If record = "Agent State End"
 begin
   AgentStateComplete = true;
 end
 /* if the record is a link to the latest complete agent state */
 Else if record = "Agent State link ADDR" AND  AgentStateComplete = true
 begin
   CompleteAgentStateLink =  ADDR;
 end
 Else If record = "MID END"
 begin
   CommitmentsComplete[completecounter++] = MID;
end
 Else If record = "MID UNDO"
 begin
   CommitmentsUndone[undonecounter++] = MID;
 end
Else If record = "MID START"
 begin
   CommitmentsActive[activecounter++] = MID;
 end
end
```

```
/* Recovering from failure*/
Procedure Recover()
begin
 /* number of commitments restored */
 restoredcounter = 0;
 /* Get the complete agent state from the Agent State Log present at the link 'CompleteAgentStateLink' */
AgentState = GetAgentState(CompleteAgentStateLink);
/* restore the state of each commitment in the agent state */
For each <MID, {placesActive}, {(variableName, variableValue)}, taskDataLink> in AgentState, not in
CommitmentsUndone
begin
   RestoreTask (MID, placeActive,Variables,taskDataLink);
   RestoredCommitments[restoredcounter++] = MID;
   /* for commitments completed complete the restored task to bring it upto date with it's state before failure  */
   If MID in CommitmentsCompleted
   begin
     CompleteTask(MID);
   end
  end
 /* get the active commitments not yet restored, that is,  commitments active but not present in the last state recorded */
 CommitmentsNotRestored =  GetCommitments(CommitmentsActive, RestoredCommitments);
 /* restart the tasks for commitments not yet restored */
 StartTask(CommitmentsNotRestored);
End
```

If a withdraw message is received in any state or for an atomic task a critical action fails, the agent writes the record <MID undo> and moves to the end place P7. For withdrawn tasks or tasks aborted due to the failure of one of the actions in an atomic task, it releases the resources and discards the places, variables and data corresponding to the commitment. If undo is not possible (that is, no withdrawal of commitment is allowed and atomicity is not required) the agent can reflect the changes in the environment and the internal state (mental state) of the agent in their place from the log before the completion of the task. Otherwise, only for completed tasks the changes from the log are reflected in their place in stable storage. Then the resources reserved for the task are released.

4.4 Recovery Protocol

The recovery algorithm for an agent to recover from failure using the ‘Commitments Log’, ‘Agent States Log’ and ‘Message Pool’ is given in **Table 2**. The last complete state is used here but partially complete state can be used to get the state of commitments for which the state was completely recorded in the partially recorded state as given in Table 3. So, even partly written agent states are useful during recovery by helping in minimizing redo of actions.

Search Agent Example: For Figure 2, assume the logs as given in the corresponding search agent Example in the logging section, section 4.3. First, ‘agent state log’ is checked. The records in ‘agent state log’ are

<2, {P5a, P5b}, {(ActionNumA, 2), (ActionNumB, 2)}, DataLink>; <3, {P5a, P5b}, {(ActionNumA, 2), (ActionNumB, 1)}, DataLink>

The agent restores the Petrinet state with places P5a and P5b active for the commitment with MID 2. Both P5a and P5b start with the second action of searching because the first action is complete. The link to data is used to get the database handlers. The actions for this commitment are continued. The agent now restores the Petrinet state with places P5a and P5b active for commitment with MID 3. For P5a the agent starts with the second action since it’s ActionNum is 2 and for P5b start with the first action. The link to data is used to get the handler to database 1. The commitment of searching is continued. The commitments log is now seen. It has the records: <agent state start>, <4 commitmentmade>, <agent state end>. It is seen that a commitment was made for MID 4 but the task corresponding to this task has not started. The task corresponding to this commitment is started. The input for this is taken from the message which is in the message pool. An old message is seen with a new MID, 5, in the message pool. This is processed as though this message has just arrived. With this the recovery is complete. For each commitment, a check can be made with the sending agent whether the commitment has been withdrawn. If it has been withdrawn, the record <MID undo> can be written in the ‘commitments log’. If the agent fails during recovery, we can redo recovery in the same way.

4.4.1 Completeness of Recovery Protocol

The agent can recover from failure in all the states and transitions of the Petrinet as shown in Table 3. So, recovery is complete**.**

4.4.2. Minimal Logging

Since our aim is to do recovery in case of agent failure with less redo of work our logging is not the minimum required for agent recovery. Removal of each log record that is being used is considered here. If we remove <MID Commitmentmade>, we may later on not commit to this, though we have committed to this now. So, this is essential. If we are sure that we will commit to this even later then we need not log this. This log record will also be useful in reducing the work to be redone. <MID End> record is required to indicate that this commitment has completed and cannot fail now. We need to write this to indicate that the commitment cannot be withdrawn now and the changes of this commitment can be made permanent.

Table 3: Agent Recovery

Kind of Failure	**Recovery**
If the agent fails after or before receiving the message. (Fails in the transitions T10 or T9 (in receiving agent) or in places P1 or P0)	It starts again behaving as if it did not receive the message. The message is taken from the message pool again.
If rid is not zero (this is a response to an earlier message) and it fails before <MID end> is written in the 'commitments log'. (Fails in the transition T2)	The message is sent to the task, which originally sent the message to which this is a response. If this message was received earlier it would be discarded by the task.
If rid is zero and it fails before making the commitment or before sending the negative response to the sender (in case the agent does not have the capability) for the task. <MID commitmentmade>, <MID end> and <MID undo> are not present in the 'commitments log' (Fails in the transitions T1 or T3 or T4 or T11, or in places P2 or P3)	The agent starts from the place P1 itself behaving as if it received the message a new.
If the agent does not have the capability to do the task, that is, <MID end> is present and <MID commitmentmade> is not present in the 'commitments log'. (Fails after T11 in the P7 place).	Since the negative response is sent to the sender of commitment before writing the log record <MID end> nothing is to be done for the recovery here.
If commitment is made but not satisfied i.e. the log record <MID commitmentmade> is present but the log records <MID end> and <MID undo> are not present in the 'commitments log'. (Fails in transitions T5 or T6 or T7 or T8 or T9 (in sending agent) or in places P4 or P5 (that is, Petrinet in place of P5) or P6)	For each such MID, the agent finds the state of the commitment in the latest agent state which has this commitment's complete state from the 'agent state log' and restores the state of the commitment to this state and continues. In case there is no entry for this commitment in the agent state, the agent assumes the task for this commitment has not started and starts the task going to P5 place.
If the commitment has been satisfied, that is after writing <MID commitmentmade> and <MID end> in the commitments log the agent fails (after transition T8 in place P7)	For each such MID, the agent can continue the task from the last agent state and complete it. At the next check point all changes of the task (both in the environment and internal to the agent) will be in the 'the agent state' log. From the log the changes can then be reflected in their place in the stable storage.
If the commitment has been withdrawn or if the task is atomic and an action has failed, that is the log record <MID undo> is present in the commitments log.	The agent drops all such MIDs as though they never started
If the agent fails while writing the state of the agent into stable storage. That is, <agentstatestart> log record is present but <agentstateend> log record is not present in the 'commitments log'.	The latest agent state will be partly complete i.e. some commitments will be written into the 'agent state log' while the others will not be written in the last agent state. The commitments for which the state was recorded completely will take this to be their latest state and the commitments for which the state was not recorded completely will refer to the older state and continue from there. We also complete the partially written agent state in the 'agent state log'. When we are finished we write to the 'commitments log' <agentstateend>. Discard the old state now.

If <MID Undo> record is not written, after recovery we may still continue with the commitment even though it has been withdrawn. If we do not write the agent state at checkpoint, we need to restart all the incomplete commitments.

This increases the redo work particularly when the task is very long. We also need the state logging of completed commitments to ensure idempotence of changes made by the commitment.

5. RECOVERY IN A MULTI-AGENT SYSTEM

Since agents are autonomous, multi-agent systems are mostly distributed and the tasks in the multi-agent systems are hierarchical the logs are distributed. Each agent has logging corresponding to the tasks done in the agent as in a single-agent system. The agent at which the task starts is the coordinator for the task, that is, the task at the coordinator is not a sub-task of any other agent's task.

If atomicity is not required:

The *agent may not wait for it's sub-tasks to complete:* No changes are needed in recovery in multi-agent systems.

The agent waits for it's sub-tasks to complete: The recovery need not access agents which have completed the task and reported the success / failure to the parent. At recovery time, the recovered agent should ask all it's sub-tasks who have not reported their success / failure. The agent reports to it's parent the success / failure of the assigned task if it hasn't reported already. Note even if some sub-tasks fail the task may succeed (example: OR of two sub-tasks). ***If atomicity is required*** then all agents involved in the task should either commit together or abort together. We need a commit protocol.

5.1 Commit protocol for a multi-agent system

In distributed database systems or nested transactions we know all the sub-tasks involved in a transaction at the start of the transaction. In distributed agent system, we don't know at the start of the task all its sub-tasks. At the coordinator agent, we only know its immediate sub-tasks. The sub-tasks in other agents may involve further sub-tasks being done by some other agents. Some of these agents may be human like a person dispatching a book. A human agent is treated similar to other agents involved. The human agent has to send or receive vote commit or vote abort for requests sent or received. Another option is to have an interface agent for humans which does the interaction with the rest of the multi-agent system for the human. The centralized solution is not preferable because of the distributed nature of most multi-agent systems with each agent being autonomous. Since each agent involved only knows about it's parent and it's immediate children because of hierarchical delegation of tasks, and the agents are autonomous a linear 2PC like protocol is a natural choice. The solution is Linear 2PC with modifications as described below.

Linear Distributed Solution**:** Here the agent sending the message to other agents waits for the other agent either at the call point or the commit point. The coordinator assigns tasks to other agents and goes into WAIT state. The first change is that if the agent carrying out a sub-task has further sub-tasks it assigns the sub-task and also informs it's parent so that the parent can increase it's timeout in the WAIT state. Now this agent goes into WAIT state. Each addition of agent is informed all the way up to the coordinator so that timeout can be adjusted in each agent. If an agent has no further sub-tasks for other agents, the agent sends either Vote commit / Vote abort to it's parent. In case of Vote commit the agent then goes into READY state. The agent aborts in case of Vote abort after sending Global Abort to it's sub-tasks if it has any. Another difference from 2PC is that when an agent receives a Vote abort from it's sub-task or timeout occurs in WAIT state, the agent may decide to abort only if the sub-task is critical and send Vote abort to it's parent and Global abort to all it's sub-tasks. If the sub-task is not critical, it may decide Vote commit / Vote abort based on other actions (for Example: if a task is the OR of two sub-tasks, then even if one fails, we can send Vote commit to the parent if the other succeeds). If the coordinator decides to commit based on the vote commit / vote abort sent by it's sub-tasks, it sends Global commit to it's sub-tasks which are to be committed for the task of the coordinator to complete and Global abort to others. The coordinator then goes into COMMIT state and commits the changes of the task in it's log by writing <MID end> in it's log (for example, if the task is OR of two tasks, if both send vote commit, the coordinator decides to commit and sends Global commit to one and Global abort to another). The Global commit receiving agent forwards Global commit to its critical sub-tasks. This agent then commits the changes of it's task by writing <MID end> in it's log. This agent finally sends an acknowledgement to it's parent. The parent agents involved in a task including the coordinator cannot remove the record <MID end> from their log until they have an acknowledgement from all their sub-tasks to which they sent a Global commit. If abort has to be done, the <MID undo> record is written in the log. We can have the presumed abort method (as for 2PC) in which for commitments aborted the agent does not wait for the acknowledgements to remove the record <MID undo> from it's commitments log. If a child fails during recovery it checks with the parent for the task using the MID. If there is no entry for this MID in the parent the child can assume that the task was aborted. The recovery and termination protocols remain the same as in Linear 2PC [11]**.** If a task is aborted, all it's sub-tasks should be aborted. But the parent tasks may or may not be aborted depending on how important the aborted sub-task in the child is to the parent. We can set fixed timeout for each agent in the WAIT state by assuming that the number of involved agents is maximum.

The Centralized solution**:** To save the number of messages 2PC may also be used in case of broadcasting networks. Here whenever a new agent is involved it's address is sent to the coordinator and the coordinator address is sent to the new agent. The coordinator gets from each agent Vote commit / Vote Abort / Vote undecided message. For Vote undecided the agent also sends the Petrinet (the task part of the Petrinet (P5 place) or a new Petrinet showing the requirements of the other sub-tasks, whether it is OR of sub-tasks or AND, etc.) of the sub-task this agent is doing. For vote abort the agent sends Global abort to all it's sub-tasks. For vote commit the agent goes into WAIT state. For Vote undecided message, the coordinator decides whether the vote is commit or abort based on votes of other agents involved in the sub-task and their dependencies. (For example an agent has the task of an action and a sub-task. If the agent completes it's action it sends to the coordinator the part of the Petrinet showing the other sub-task as compulsorily must be done. The coordinator decides this is vote commit if the other agent sends vote commit. If the other agent sends vote abort it decides this is vote abort. If the other agent sends vote undecided votes of it's sub-tasks are used to get the vote). The coordinator then decides that all agents should abort (Global Abort case, a global abort is broadcasted to all participating agents) or some should commit and some should abort (Global Commit case, two multicast messages one for Global commit and one for Global abort are sent to the appropriate agents) based on the Petrinets and the votes. Acknowledgement of commit / abort is sent by the receiving agents. The recovery and termination protocols remain the same as in 2PC. Note that this wouldn't work if an agent waits for it's sub-task to complete at call point and then does it's other actions. This has the disadvantage of being centralized which is against the hierarchical delegation of tasks in multi-agent systems. This is also against the nature of distributed multi-agent systems. Also the autonomy and security of agents may be compromised here. Sometimes there may be links only between parent and child in which case the centralized solution cannot be applied.

6. CONCLUSIONS AND FUTURE WORK

We have used a Petrinet to model an agent. We have handled the problems of durability and atomicity in an agent when failures occur. We have proposed an algorithm to generate the log records. We have proposed a recovery algorithm using these log records and shown that the recovery is complete. The proposed logging is to be done in each agent of the multi-agent system. For atomicity of tasks in multi-agent systems linear 2PC with some modifications is proposed. This is a theoretical work; the implementation is yet to be done. The granularity of reservation (locking) and the modes of reservation (like read and write locks) are being currently worked on. For consistency, the changes of a task can be checked for consistency violations (like contradicting beliefs) before committing and the task can be aborted if any violations do occur.

7. ACKNOWLEDGEMENTS

We are thankful to Dr. Garimella Uma for the valuable suggestions she gave us while doing this work, particularly in modeling the agent as a Petrinet.

REFERENCES

[1] Q Chen, U Dayal, "Multi Agent cooperative transactions for E-Commerce", Cooperative Information Systems, 7th International Conference, CoopIS, 311-322, 2000.

[2] R Elmasri, S B Navathe, Page 640, Section 19.3, "Fundamentals of Database Systems", 3rd Edition, Addison Wesley.

[3] K Jensen, "Colored Petrinets", book published by Springer Verlag, New York, 1995.

[4] D Moldt, F Winberg, "Multi Agent Systems based on colored Petrinets", 18th International Conference, ICATPN, France, 82-101, 1997.

[5] F Brazier, B Keplicz, J Treur, R Verbrugge, "Modeling internal dynamic behavior of BDI Agents", Third International workshop on Formal Models of Agents, Modelage Workshop, Springer Verlag, 36-56, 1997.

[6] Y Shoham, "Agent Oriented Programming", AI 60, 51-92, 1993.

[7] Q Chen, U Dayal, "Failure handling for Transaction Hierarchies", Proc. Of 13th International conference on Data Engineering (ICDE-97), UK, 245-254, 1997.

[8] J Gray, P McJones, M Blasgen, B Lindsay, R Lorie, T Price, G Putzolu, I Traiger, "The Recovery Manager of the System R Database Manager", ACM Computing Surveys 13, 223-243, June 1981.

[9] O Herzog, W Reisig and R Valk, "Agents and Petrinets", Petrinet Newsletters, number 49 in Petrinet Newsletters, 1995.

[10] S Burkhard, "How to define properties – or: What is a fair agent?" Technical report, Fachbereich Informatik, Humboldt-Universitat Berlin, 1993.

[11] J Gray, Notes on Database Operating systems in Bayer, Graham, Seegmuller, editors, Operating Systems: An Advanced Course, New York; Springer Verlag, 1979.

[12] M Klein, C Dellarocas, Exception Handling in Agent Systems, Proc. of the 3rd Annual Conference on Autonomous Agents, Seattle, 62-68, 1999.

[13] S Haegg, A sentinel approach to Fault-handling in Multi-Agent Systems, Proc. of the 2nd Australian Workshop on Distributed AI, Queensland, Australia, 181-195, 1996.