Consistent and Recoverable Agent-Based Access to Heterogeneous Mobile Databases

Evaggelia Pitoura

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Report Number:
95-011
CONSISTENT AND RECOVERABLE
AGENT-BASED ACCESS TO
HETEROGENOUS MOBILE DATABASES

Evaggelia Pitoura
Bharat Bhargava

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR-95-011
February 1995
Consistent and Recoverable Agent-Based Access to Heterogeneous Mobile Databases *

Evaggelia Pitoura and Bharat Bhargava
Department of Computer Science
Purdue University
West Lafayette, IN 47907-1398
{pitoura, bb}@cs.purdue.edu

Report CSD-TR-95-011

Abstract

Today's information applications have a functional need to distribute themselves among numerous remote sites through both wireless and wired links. To overcome the communication barrier and support the development of complex applications, traditional models of distributed computing need to be revised. In this paper, we advocate an approach based on agents, which are software modules that encapsulate data and code, cooperate to solve complicated tasks and run at remote sites with minimum interaction with the user. We define an agent-based model for accessing mobile heterogeneous databases. We then investigate concurrency control and recovery issues and outline possible solutions. Agent-based computing advances database transaction and control flow management concepts and remote programming techniques.

Keywords: agents, mobile computing, flow of control, multidatabases, transactions, concurrency control, recovery

1 Introduction

With the rapid development of networking, computing applications increasingly rely on the network to obtain and update information from numerous remote sites. These application will often

*work supported in part by a Purdue Research Foundation Grant
have to use wireless connections. A serious restriction of mobile computing is that users are only intermittently connected with the rest of the distributed system [15, 22]. Wireless connections are expensive, unreliable, and slow. In addition, most portable devices have limited computing resources.

Today, networking is based on remote procedure calling (RPC), where a network carries messages. To overcome the communication barrier, there is a need for lighter-weight and more flexible architectures. A different approach is remote programming (RP) [30]. In this case, a network carries agents which are objects that encapsulate data and procedures that the receiving computer executes. To execute an agent is to perform its procedures in the context of its data. Agents make the necessary distribution logistically simpler and overcome the communication barrier by minimizing the number of interchanged messages. Furthermore, agents can be easily customized, thus making the network an open platform for developers.

There is a lavish of research on agent-based systems [25] accompanied by new commercial products. Among them, General Magic's Telescript is being used in AT & T's PersonalLink servers and in Sony's Magic Link [32, 30]. Most research on agent-based models focuses on aspects related to intelligence. Although some of the intelligent characteristics attributed to agents are far from reality, there is a short-term product potential in agent technologies especially for database agents, email filters, and group enabled applications [13].

In this paper, we focus on consistency and recovery aspects of agent-based computing. We provide a framework for agent-based access to heterogeneous mobile databases, lay down its implications and identify its differences from traditional database models. We then introduce appropriate workflow constructs and outline solutions for concurrency and recovery.

The remainder of this paper is organized as follows. The agent-based model of computing is introduced in Section 2, along with the identification of its implications on concurrency control and recovery. In Section 3, we present a control flow model for agent-based computing. In Section 4, we discuss concurrency control and in Section 5 recovery. In Section 6, we compare our work with
related research. Finally, in Section 7 we offer conclusions.

2 The Agent-Based Model

In an agent-based computing system, applications that want to interact with database systems submit agents, which in this paper are called application agents or simply agents. Agents, like distributed processes, model task executions. Multiple agents are being executed concurrently accessing one or more database systems.

Each agent is written in a high-level distributed programming language that allows parallelism inside the agent. The structure of the agent is then translated to a set of dependencies among execution states. This information is used to schedule the internal parts of an agent. Additional dependencies may result from concurrency control to ensure database consistency.

To support interoperability, the heterogeneous database systems that participate in the agent-based architecture export a predefined set of operations, called primitive (database) methods. Agents can access these heterogeneous systems only through the provided primitive methods. Furthermore, in each database system, special agents, called database agents, coordinate the access to local data items. The database agents are responsible for maintaining the consistency of local systems and providing recovery in case of failures. After the submission of a primitive method the application agent has no further control on that method.

In addition, an application agent can communicate with other application agents to cooperate in the execution of their assigned tasks. This communication is accomplished by invoking methods on the other agent’s local data. Again, these operations are selected from a pre-specified set of primitive methods called primitive application methods.

Specifically, an agent is an active object. It consists of local data, methods, and dependency specifications among methods. An agent executes its methods in the context of its data. A method is: (a) a predefined primitive method that accesses some local database or some other agent’s local data, (b) a local method on the agent’s data, or (c) some combination of local and primitive
methods. Methods can modify an agent's local or remote data. In addition, methods may result in modification of the agent's own specifications.

2.1 Examples

We present three agent-based scenarios. The first describes electronic news filtering and is slightly adapted from [17]. The second is from the electronic market place and is slightly adapted from [31]. Finally, the last example is about mobile computing and is taken from [15].

Example 1: Electronic News Filtering. Mary creates an application agent whose responsibility is to select from the available news those that match her interests. Mary's interests are specified in a special profile database. The agent performs some form of text analysis to retrieve words in the news that match those specified in the profile. Different agents may be created to express Mary's different interests, for instance a "politics agent" searches news about politics while a "literature agent" searches news about new book releases. The agents take feedback from Mary and then update her database profile to reflect her current preferences.

Example 2: Shopping. John is searching for a new camera. He creates an application agent with specific instructions to find the cheapest camera of a specific brand and purchase it using John's credit card account. John's agent first visits the yellow pages, and then each shop listed. The application agent, after negotiating with each database agent to find out the price, returns to the database agent of the shop that offers the best value and purchases the camera. Finally, the agent updates John's personal files by talking to the database agent that supervises them.

Example 3: Mobile Computing Mary, an insurance agent, while traveling to a prospective customer creates an application agent to check the customer consumer's record and other credentials. The application agent is executing even when Mary turns off her palmtop to save energy. Meanwhile, the application agent may interact with other application agents, for instance by a company's agent, to find out what would be an appropriate offer to that customer.
2.2 The particular characteristics of the model

We identify the following characteristics of agent-based computation that have an impact on concurrency control and recovery.

1. There are similarities to concurrency control in multidatabase system, since an agent access multiple heterogeneous, autonomous, and distributed database sites. However, there are some important variations from traditional multidatabase schemas:
   - Agent-based computation is decentralized. There is no global transaction manager or central database agent, since the environment is open and evolving. Agents can be submitted from various sites including mobile stations. Having to route all agents through a central point is both unrealistic and inefficient.
   - The decomposition of an agent into local and primitive methods is not known at the time of its creation since it depends on the result of the execution of its previous actions and can be dynamically modified during run-time.
   - The interface offered by each local database system is a collection of primitive methods in contrary to the common assumption in multidatabase concurrency control, where applications interact with the local systems by submitting read and write operations.

2. Agents need to coordinate with each other and exchange information to collectively solve a problem. Thus, each agent cannot be executed as an isolated transaction. In addition, since agents execute complex activities there is a need for extending facilities to support advanced control flow features.

3. Each agent has each own local data.

4. Each agent is a robust, recoverable object. In case of a failure, not only the data items in the databases should be restored but also the local data and the computation state of the agent.
Figure 1: (a) Traditional multidatabase architecture (b) Agent-based architecture

In Figure 1, the architectural differences between traditional multidatabase systems and agent-based systems are highlighted.

3 Flow of Control Specification

An application agent is created to execute a task. Agents have a structure determined by control flow specifications similar to that of ordinary programming languages. In addition, agents can communicate with other agents based on an agent communication language and a set of primitive methods.

In terms of primitive methods, the finest granularity of the lifetime of an agent, that can be controlled at the time of its specification or execution, is the completion (commit, or abort state) and the submission (begin state) of the method. The actual execution time of a primitive method is under the control of either the local database agent or the corresponding cooperating...
agent. In addition, some local database agents provide a prepare-to-commit state, that indicates that a primitive method has completed execution and the results are about to become permanent. In terms of local methods, an agent has also control over their time of execution. We distinguish two types of commitment depending on the result, namely semantic failure and semantic success.

In summary, the controllable states in the lifetime of an agent are: commit (semantic failure and semantic success), abort, prepare-to commit, and submit a primitive or local method and execute a local method.

3.1 Structural dependencies

Structural dependencies are dependencies among controllable states of the methods of an agent. We distinguish three types of structural dependencies, trigger, ordering, and real-time dependencies. Let $M_i$ be a method and $s_j$ be a controllable state.

- **Trigger dependencies** are of the following form: if $M_1$ enters state $s_1$ then $M_2$ must enter state $s_2$. Special cases of structural dependencies include critical, contingency, and compensation methods. Critical methods are methods that when aborted (or semantically fail) the whole agent must abort (or semantically fail). Contingency methods are methods that are executed as alternatives, when a method fails semantically. Compensation methods are methods that are executed to semantically undo the effect of a committed method when some other method aborts.

- **Ordering dependencies** have one of the following form: $M_1$ can enter state $s_1$ only after $M_2$ has entered state $s_2$, or $M_1$ cannot enter state $s_1$ after $M_2$ has entered state $s_2$. They can be used to express data flow dependencies, for instance the fact that $M_1$ reads data produced by $M_2$.

- Finally, language constructs can be used to express real-time dependencies based on requirements about the submission and completion time of a method.
All methods of an agent can be executed in parallel, unless otherwise indicated by an ordering dependency or by restrictions imposed by concurrency control. The ordering dictated by concurrency control is necessary for the consistency of the database data or the local data of an agent.

3.2 Cooperation among agents

Supporting cooperation among application agents is an important characteristic of the agent-based model since agents are by nature synergistic [12].

To support agent cooperation, General Magic, for instance, provides a command called meet, by whose invocation two agents can interchange information, if so authorized by their specifications [32]. The Knowledge Query and Manipulation Language (KQML), being developed as part of a large DARPA-sponsored knowledge initiative, is a language and a protocol to support the high level communication among intelligent agents [10]. In KQML, agents communicate by exchanging messages.

Thus, agents, in contrast to traditional transactions, are not isolated from other concurrently executing agents. To formalize this principle, one can borrow concepts from advanced transaction models. An appropriate such concept is the concept of a breakpoint [9]. According to this approach, two transactions (agents in our case) can interchange operations only at pre-specified points of their executions, which are called breakpoints.

3.3 Relocatable applications

In the remote procedure calling model, programs are immobile and each application is statically installed on a site. In agent-based programming, however, agents dynamically move to different sites. The property of having the center of computation move is very important in a computing environment with wireless connections.

In a wireless computing environment the location of a user changes dynamically with time. Thus, the distance of a client from an information provider is not a fixed parameter of the cost of
the service. Consequently, relocating a computation is essential for minimizing the communication cost and improving the response time by minimizing the physical distance between information providers or by taking into consideration the changing network load and availability. The mobility of agents is also useful for balancing the load among base stations or in cases of network or server failures.

4 Concurrency Control

Multiple agents are concurrently accessing shared resources. There are two aspects of correctness. One aspect is maintaining the consistency of shared resources, which is usually specified through a number of integrity constraints. We call this form of correctness data correctness. The other is ensuring the structural properties of agents as specified by the flow of control constructs. We refer to this form of correctness as structural correctness.

4.1 The model

The execution of an (application) agent consists of local steps, primitive message steps and composite message steps. A local step is the execution of an operation on an agent’s local data. A primitive message step is the execution of one of the primitive methods provided by a local system or by another application agent. A composite message step is the execution of a composite message and consists of a number of primitive and local steps. Finally, an agent execution is a partial order \((T, \succ)\) where \(T\) is a set of local, primitive and composite steps, and \(\succ\) is an order on the steps determined by the algorithmic structure of the method and its structural specifications.

At each local database, in addition to the application agents, a number of local applications are also being executed. We call them autonomous-local messages or autonomous messages for brevity. We assume that each primitive message step offered by a local site or another agent is executed as an atomic, isolated, local-site or agent consistent, and durable transaction. Ensuring these properties is the responsibility of the local site or the agent respectively.
Structural correctness

Structural correctness refers to the maintenance of the control flow characterization of each agent. We assume that, as in TeleScript [32], there is an engine (an interpreter) as part of the agent, that executes the local methods and controls the structural dependencies. Handling structural dependencies can be accomplished for instance by using petri nets, or active rules.

Data correctness

Data correctness refers to the maintenance of the consistency of the data participating in the computation. In the agent-based model, maintaining consistency refers also to maintaining the consistency of the local data of each agent, since parts of the code of an agent may be executed concurrently, and since other cooperative agents may access the agent’s local data.

The prevailing approach to maintaining data correctness and ensuring the isolation property of an execution is serializability. Each local database site ensures the serializability of all methods at its site. We also assume that each agent ensures the serializability of all methods (local and primitive) being executed on its data. Then, global serializability refers to finding a serialization order consistent with the serialization orders assumed by all local database sites and all agents.

There are various aspects that determine the difficulty of this task, including:

1. the existence of autonomous local messages. Autonomous messages are messages submitted by agents that are executed outside of the control of the database agents. These local agents are part of the autonomous local systems and are completely hidden from the (application) agent.

2. the existence of interdatabase constraints, which are constraints that span more than one local database sites. Constraints may also exist among local data of different agents or among local data of an agent and a local database. In the absence of these kinds of constraints, there is no need for global serializability to maintain data correctness. However, some means of serializability may be necessary to ensure some form of agent isolation.
3. the type of local schedules produced by local database agents. For example for strict schedules, e.g., schedules whose commitment order is the same as their execution order, enforcing global serializability is easier.

We outline a timestamp-based method to ensure global serializability. The proposed method takes advantage of the fact that the interface offered by each system is a repertoire of methods. A commutativity relation is defined for each set of primitive and local methods. Two methods $M_1$ and $M_2$ commute if they do not conflict, that is, if the result of executing $M_1$ after $M_2$ is the same as executing $M_2$ after $M_1$. These relations are saved in the form of a commutativity matrix.

In a closed-nested transaction model, such as that presented in [14], conflicts among primitive or local transactions will result in conflicts among any composite methods that invokes them. In open-nested transactions [19] there is no such implication.

A decentralized concurrency control algorithm. The algorithm is a variation of [2] for high-level operations. The algorithm is for open nested transactions but can be adopted for non-open nested (and thus provide less concurrency) using, for instance, hierarchical timestamps [14]. Each agent gets a timestamp when created. The timestamp is defined to be a combination of the value of the clock and the user’s id. The timestamp of an agent will correspond to the global serialization order of the agent.

Each agent serializes all conflicting local operations on its local data based on the timestamp order. An operation on its local data issued by another agent is executed only after ensuring that the two agents are allowed to “meet”.

We now describe the submission of a primitive method from an (application) agent to a local database agent. Note, that each agent in order to execute a composite method can use, for instance, the semantic-based locks of [19].

Each local database agent has a variable, called agent ticket ($AT$). In the case of autonomous messages, an additional data item per database site is needed. This data item is physically stored in that site and is called a physical ticket ($PT$).
Each local agent keeps a list of the timestamps of all primitive methods that have been submitted to the site. A method that does not commute with a submitted method is not allowed to execute concurrently with that, thus if such a method arrives with timestamp smaller than \( AT \) is aborted. Two commutable methods can be executed concurrently. However, to avoid indirect conflicts, that is conflicts imposed by local autonomous methods, we want to force them to conflict directly,

**Submitting a commutable method \( op \)**

\[
\text{get}(AT) \\
\text{if} \ (AT > \text{method timestamp}) \\
\quad \text{abort the method} \\
\text{else} \\
\quad \text{submit}(op) \\
\text{then in a critical region} \\
\text{get}(AT) \\
\text{if} \ (AT > \text{method timestamp}) \\
\quad \text{abort the method} \\
\text{else} \\
\quad \text{write}(PT, \text{method timestamp}) \\
\quad \text{send prepare-to-commit to agent} \\
\quad \text{if decision taken to commit operation} \\
\quad \quad \text{set} \ (AT, \text{method timestamp}) \\
\quad \text{else} \\
\quad \quad \text{abort method}
\]

If there are no local autonomous methods, then commutable methods are allowed to execute concurrently without any additional control.

**Consistency Annotations.** An alternative method for concurrency control is based on anno-
tating methods with consistency assertions. Specifically, for each primitive or local method, an assertion, called a \textit{precondition}, expresses the input requirements, and a second assertion, called a \textit{postcondition}, expresses the properties ensured by any call to the method. Formally,

Methods are defined as \textit{Hoare triples}. Each method $\text{op}_i(a, b)$ on an object $o$, where $a$ is an input parameter, and $b$ is an output parameter, is annotated as follows: \{(P_i) \text{ op}_i(a, b) \} \{Q_i\}, where \{P_i\} \equiv \text{Precondition}_i(o)$ and $\{Q_i\} \equiv \text{Postcondition}_i(o)$. $\text{Precondition}_i(o)$ and $\text{Postcondition}_i(o)$ are predicates on $o$’s local state and on $a$ and $b$.

Again the only assumption is, that primitive methods are implemented atomically at each local site or at each cooperating agent, and in a manner that preserves the consistency of the local site or agent. Any call to a primitive method may assume that the invariant is initially satisfied (i.e., it may expect to find data in a consistent state), but it must maintain the invariant (i.e., it leaves data in a consistent state). Consistency annotations offer a natural and convenient formalism for incorporating the consistency requirements of data into their interface.

Consistency assertion can be used to define when two methods conflict [21]. In the following, we illustrate how some proposed definitions of commutativity can be expressed using consistency assertions. $wp.\text{op}_j. P_i$ stands for the weakest precondition for $P_i$ to hold after the execution of $\text{op}_j$. The list of definitions is intended to be illustrative rather than exclusive. Two basic methods \{P_i\} $\text{op}_i \{Q_i\}$, and \{P_j\} $\text{op}_j \{Q_j\}$ commute if:

- $P_i \land P_j \Rightarrow wp.\text{op}_j. P_i$ (non-interference) [20];
- $P_j \land wp.\text{op}_i. P_j \Rightarrow P_i \land wp.\text{op}_j. P_i$ (left commutativity) [1, 14];
- In any state where both $P_i$ and $P_j$ hold (both $\text{op}_i$ and $\text{op}_j$ can be executed), $P_i \Rightarrow wp.\text{op}_i. P_j$, and $P_j \Rightarrow wp.\text{op}_j. P_i$ (forward commutativity) or, if $P_i \Rightarrow wp.\text{op}_i. P_j$, then $P_j \Rightarrow wp.\text{op}_j. P_i$ (backward commutativity) [29].

Weak consistency. Distributed systems that include wireless connections are characterized by
frequent and predictable disconnections. As a consequence, agents executed on mobile hosts usually have to rely on locally available data to perform their computations. These data may be out-of-date. An approach to handling inconsistency is to introduce weak read and weak write operations as part of the interface of a database agent to allow application agents to access local, possibly inconsistent data [24].

5 Recovery

In a distributed system, types of failures include communication, site and processing failures. In the agent-based model, the information that must be recoverable, called context, includes the local data of the agent and the environment of its computation. Following a similar approach to that of Contract [28], we assume that the context is saved in a private database for each agent called context database. The context database is stored in stable storage and survives failures.

Instead of overwriting the local data of an agent, versions are kept. The value last written to a data item by a committed operation is called the last committed value for that data item. A committed (context) database state with respect to an execution is the state in which each data item contains its last committed value. Methods of an agent that are executed concurrently at different local databases or cooperating agents take a copy of the appropriate local context database.

We rely on system support provided by local database agents for failures of local sites while an agent's primitive method is being executing. We assume that either such failures are transparent to the user or that they result in an abort of the requested method. When a site failure occurs after a primitive method was successfully executed but before its results become persistent, the method must be compensated. Then the method should be either retried or considered aborted.

Site failures that occur while a local method is executed are handled by using the local context. A committed context database state is restored.

Communication failures are hard to detect. They are usually detected by specifying time-outs. In such cases, when an agent is lost, it is reconstructed using the context.
To bound the size of the context database who keeps growing with the storage of versions, some of the old versions must be deleted. An entry can be removed if the method that wrote it has been aborted. An entry can also be deleted when overwritten by a method that commits.

5.1 The atomicity property

The atomicity property expresses the requirement that all or none of the methods of an agent is committed. Ensuring this property is complicated because each local database agent makes independent decisions on whether to commit or abort a primitive method. To resolve this problem, most systems either attempt to compensate (semantically undo) a committed method or to retry an aborted method. The atomicity property is not appropriate for agents because usually aborting the whole computation is too costly. Alternatively, characterizations have been proposed, such as contingency and compensating transactions. We do not discuss this issue any further. The interested reader is refer to [4] for an overview of some of the proposed approaches.

6 Related Work

Techniques to support an agent-based model of accessing database systems combine concepts from multidatabase concurrency control, advanced transaction models and control flow management.

Much current research on transaction management has focused on the problem of concurrency control in multidatabase systems. [5] offers an excellent survey on the topic. We have outlined the differences between agent-based concurrency control and multidatabases in Section 2.3. Multidatabase systems that offer method-based interfaces include the DOM project [18] and the VODAK system [16]. Multi-level transactions and their relation to multidatabase control is investigated in [27]. The majority of transaction management for multidatabase systems adopts a centralized approach with the possible exceptions of [33, 2].

Many researchers have identified the need for advanced transaction models. See [8] for examples. ACTA [6] provides a framework based on first-order logic for reasoning about extended transaction
models. The model is low-level. A higher-level model based on transaction primitives is described in [3]. The two models can be used respectively to express and implement some of the control flow characteristics of agents. Many researchers starting from extended transaction models have defined control flow specifications [26, 11].

Computations as recoverable objects were also discussed in [28]. Finally, transaction handling in mobile computing environments has been discussed in [7, 23, 34]. The agent-based approach is in compliance with all of these models.

7 Conclusions

A new model of distributed computing is evolving based on multiple, autonomous, and persistent communicating agents. This model is tailored to the computing requirements of the future that include wireless connections, numerous distributed sites and move the border of coordination from users to application programs.

In this paper, we have introduced a model of agent-based access to heterogeneous mobile databases. We have identified the implications of this new agent-based paradigm on concurrency control and recovery and its differences from traditional models. We have proposed control-flow constructs and outlined solution for data correctness and failure recovery. Although, we have not exhausted this new and fascinating topic, we have outlined its character and several possible solution to emerging problems.

References


