Agent Server Technology for Managing Millions of Agents

Gaku Yamamoto

IBM Research, Tokyo Research Laboratory,
1623-14, Shimo-tsuruma, Yamato-shi,
Kanagawa-ken 242, Japan
+81-462-73-4639
yamamoto@jp.ibm.com

Abstract. In this paper, we describe technologies for an agent server capable of hosting millions of agents. The agent server needs a thread management mechanism, a memory management mechanism, and a recovery management mechanism. We have developed a framework and agent execution environment named Caribbean. First, we describe the programming model of Caribbean. Following the description, we explain technologies for managing millions of agents. Some application scenarios of real commercial systems using the technology are also introduced. We describe what we learned from the development of the real applications.

1 Introduction

We used a multiagent programming model for a real commercial system in 1998. The application is a service that provides information on airline tickets to consumers through the Internet. In this service, consumers have their own computer agents. Consumers input their query conditions on flights. In the system, travel agencies also have their own computer agents providing airline ticket information. A travel agency's agent has the agency's own sales policies. When a consumer inputs his or her query conditions, the consumer's computer agent sends a query condition message to the travel agency's computer agents. A travel agency's agent replies with a message containing airline ticket recommendations in accordance with its sales policies. The consumer's agent shows the airline ticket recommendations from the travel agency's agents using a Web browser. The agent also retains the input query conditions and the airline ticket information, and can show the information anytime when the consumer uses the agent again. The lifetime of a consumer agent is one week.

Through the development of the system, we found that the multiagent programming model is flexible, and it is easy to design and develop a system. However, we had to solve the serious problem that there was no multiagent platform that managed a very large number of agents. Since each consumer has an agent in the system and each agent lives in the system for a week in the commercial system, the number of consumer agents may be in the tens of thousands. Therefore, we developed our first multiagent platform that manages tens of thousands of agents on top of the

T. Ishida, L. Gasser, and H. Nakashima (Eds.): MMAS 2004, LNAI 3446, pp. 1 – 12, 2005.

Aglets framework, a mobile agent framework [1]. In 2000, we redesigned the platform and developed a new multiagent platform named Caribbean, which can manage hundreds of thousands of agents on a single platform [2-9]. In 2003, we added a server clustering function to Caribbean so that millions of agents can be managed in one system.

In this paper, we describe the technologies for a multiagent platform capable of hosting millions of agents. We call the multiagent platform that manages many agents the "agent server" in this paper. In Section 2, we describe the programming model of Caribbean. Section 3 describes the runtime structure of a Caribbean agent server. A server clustering mechanism of Caribbean is introduced in this section. Several application scenarios will be introduced in Section 4. The lessons learned from developing real applications will be described in Section 5. Conclusion is written in Section 6.

2 Caribbean Programming Model and Framework

2.1 Programming Model

In the Caribbean programming model, an agent is responsible for given roles and performs its tasks to meet its design objectives. In the typical application scenario shown in a later section, an agent is a proxy of a user. All agents are created in a server and stay in the server for a long time. Agents in Caribbean are reactive agents that execute jobs by receiving messages or events. This means that the agent does not own a thread. Occasionally, an agent requires a special service like a timer service. Such a service is provided by a "Service Object." A service object can own threads. An agent communicates with other agents by using asynchronous peer-to-peer messaging. Messages sent by an agent are put into the message queue of the destination agent. A Caribbean agent server creates a message queue for each agent. An agent server delivers a message stored in a queue to the destination agent at an appropriate time. When a message is delivered to an agent, a callback method of the agent's methods will be invoked. In that method, the agent performs its job and may send messages to other agents.

An agent server provides agents with fundamental functions such as agent creation, agent removal, and messaging. An agent server must host a large number of agents. If each agent owns a thread, the agent server will be overloaded. If too many agents are in memory, an agent server will also fail because of the resulting memory shortage. Therefore, an agent server must manage the activities of agents to control thread assignment and memory usage.

An agent server does not provide intelligent agent capabilities. If an application requires intelligent agent capabilities, the capabilities can be added as agent logic.

2.2 Framework

Figure 1 shows an overview of the Caribbean framework.

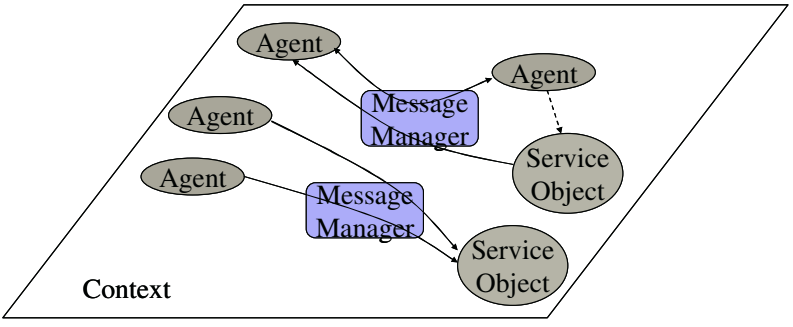

Fig. 1. Caribbean Framework

ObjectBase. The base class of agents is the "ObjectBase" class. An agent is an instance of a class extended from this class. An agent is identified by a unique identifier whose class is "OID." An agent can belong to an agent group. An application can define an agent group in a system configuration file. In the configuration file, properties for an agent group can be defined. An agent can obtain the properties from "Context." An agent must not deliver to other agents an object reference to itself. Instead, it delivers its own identifier. Instead of method calls, an agent sends other agents messages, which are instances of either the "Message" class or a class extended from the "Message" class. An agent program is based on a message-driven programming model. When an agent receives a message, a callback method "handleMessage" of the agent will be calledbacked. The delivered message is handled as an argument of the method. The agent executes the task corresponding to the message within a short period. An agent excepting an instance of the ServiceObject class described later must not own any threads. Because of memory limitations, an agent might be moved into storage by an agent server and loaded from storage back into memory. The methods that an agent must implement are as follows;

`public void onCreation(Object arg)`

arg: an argument of Context#create()

`public void onDisposing()`

This method is called by an agent server just before this agent is disposed.

`public void onActivation()`

This method is called by an agent server just after this agent is loaded into memory.

`public void onDeactivating()`

This method is called by an agent server just before this agent is stored into storage.

```
public boolean handleMessage(OID sender, Message msg,
MessageManager msgman)
```

sender: an agent identifier of a sender agent
msg: a message

msgman: an object reference to the message manager which the sender agent used to send the message.
This method is called to handle the message.

The following methods are provided by the "ObjectBase" class.

```
public Context getContext()
```

Get an object reference to a context.

```
public String getGroup()
```

Get the name of the group to which this agent belongs.

```
public OID getOID()
```

Get an agent identifier.

```
public Properties getProperties(OID oid)
```

oid: an agent identifier
Get properties of a group to which the agent specified by an agent identifier belongs

Context. Agents invoke use the agent server's functions provided through the "Context" interface. The interface provides methods for creating agents, disposing of agents, getting lists of agents in an agent server, etc. "Context" provides the following methods:

```
public OID create(String classname, Object arg)
```

classname: a class name of a created agent
arg: an argument passed to the onCreation method of ObjectBase.
Create an agent. Return the identifier of the created agent.

```
public OID create(String classname, String agentgroup,
Object arg)
```

classname: a class name of a created agent
agentgroup: an agent group name
arg: an argument passed to the onCreation method of ObjectBase.
Create an agent. The created agent belongs to the agent group. Return the identifier of the created agent.

```
public void dispose(OID oid)
```

oid: An agent identifier
Dispose of an agent.

```
public OID[] getAllOIDs()
```

Get an array of the identifiers of all agents in this agent server.

```
public OID[] getAllOIDs(String agentgroup)
```

group: an agent group

Get an array of the identifiers of all agents belong to the agent group. Return an array of agent identifiers.

```
public String getGroup(OID oid)
```

oid: an agent identifier
Get the name of the agent group to which an agent belongs

```
public MessageManager getMessageManager(String name)
```

name: the name of the messaging manager
Get an object reference of a messaging manager.

```
public SimpleMessageManager getSimpleMessageManager()
```

Get an object reference to the default messaging manager.

```
public ServiceObjectBase lookupService(String name)
```

name: a name of a service object
Get an object reference to the service object.

MessageManager. A MessageManager is an object that provides messaging functions to agents. It provides an asynchronous messaging function and a multicast messaging function to agents as fundamental functions. "MessageManager" class is an abstract class. The Caribbean package provides "SimpleMessageManager" class as a default message manager. An application can define an application-dependent MessageManager and can plug it into an agent server. For example, an application may need an anonymous messaging function that distributes a message to an appropriate agent in accordance with a message name. An application-dependent MessageManager is defined in a system configuration file. An agent server registers it into a message manager repository at system startup time. An agent obtains an object reference to a MessageManager from Context, and sends messages using those messaging functions. Context manages multiple MessageManager objects that are identified by names. The methods provided by "SimpleMessageManager" class are as follows;

```
public void post(OID[] oids, Message msg)
```

oids: an array of identifiers of destination agents
msg: a message
Multicast a message to destination agents.

```
public void post(OID oid, Message msg)
```

oid: an identifier of a destination agent
msg: a message
Post a message to a destination agent. This method does not wait until the destination agent handles the message.

ServiceObjectBase. A "Service Object" is the object that provides agents with services. A service object is an instance of a class extended from "ServiceObjectBase" class, a subclass of "ObjectBase" class. A service object has all the functions of agents. It can send messages to other agents. It also can receive

messages. Moreover, a service object can own threads. An agent can obtain an object reference to a service object and can call it by using a method call. A service object is associated with a service name and is defined in a system configuration file. An agent server registers the service object into the service registry of an agent server at system startup time. An agent can then obtain an object reference to the service object.

Service objects are also used for extending the Caribbean framework. For example, the Caribbean package provides the RMI gateway service object that a client program sends to an agent, the mail delivery service object that an agent uses to send a mail message to a user, the event transfer service object that an external program can use to send a Caribbean message to agents, etc.

3 Agent Server

The Caribbean agent server runtime is designed in order to manage many agents. Figure 2 shows an internal architecture of Caribbean agent server runtime. A thread management mechanism is needed to improve performance, and a memory management mechanism for limiting the amount of memory occupied by the agents is also needed. In addition, agents should be protected against system failures, and therefore a recovery management mechanism is needed. There are several different types of agents in a single application. For example, the number of user agents, which are created for each user, might be in the hundreds of thousands. On the other hand, an agent server has only one the service object (which provides DBMS access services for the user agents). Obviously, the management policies for user agents are different from those of the service agent. Therefore, Caribbean provides a function for defining management policies for each type of agent.

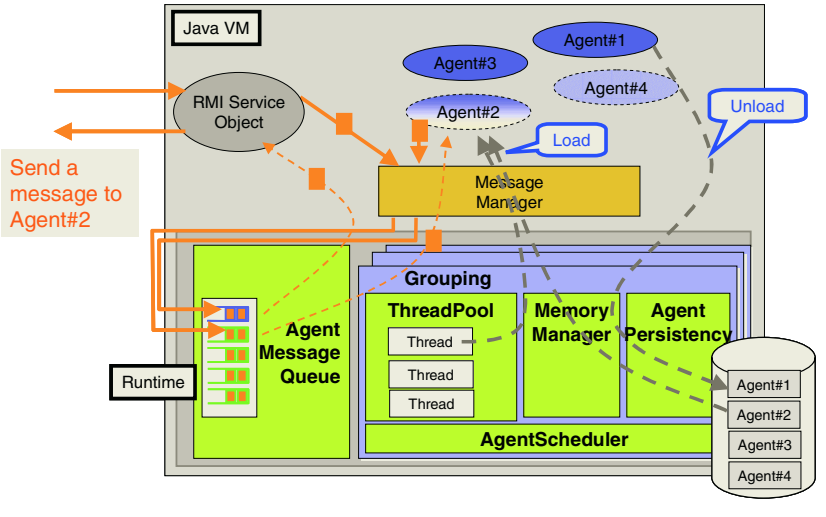

Fig. 2. Internal Architecture of Caribbean

3.1 Memory Management

The Caribbean agent server hosts hundreds of thousands of agents. An agent is a Java object whose size in a typical application may be tens of kilobytes. Even if each agent is not very big, the amount of memory occupied by all of the agents will significant, and may amount to hundreds of megabytes. Therefore the agent server needs a mechanism for minimizing the amount of memory occupied by hundreds of thousands of agents.

Agents should be kept in memory, but physical memory is limited. Therefore, the Caribbean agent server has a mechanism for swapping agents in and out. Some agents can be located in memory and others in secondary storage. If an agent in secondary storage receives messages, then the mechanism reads a memory image of the agent from the storage and activates the agent. This is called "swapping in." At the same time, the mechanism moves another agent into storage. This is called "swapping out." The agent server automatically swaps agents in and out in order to limit the number of agents in memory.

3.2 Thread Management

Since there may be hundreds of thousands of user agents, running agents without an appropriate schedule may cause overloading of the agent server. In the case of applications in which a user agent communicates with many agents accessing a database, a single request from a user may cause many simultaneous database accesses, causing an overload of the database. To avoid system overloads, a mechanism for controlling the threads is needed.

The Caribbean agent server uses thread pools and limits the number of running threads. It also has a thread scheduler in order to improve the performance. To reduce the number of swaps, the scheduler assigns higher priority to the agents in memory over the agents in secondary storage.

3.3 Recovery Management

Agents should be persistent within their lifetimes, even if the agent server fails. This means that an agent should be able to recover after the server is restarted. Agents that do not modify their data can be created afresh, but those that modify their data should be recovered together with the data. Therefore an agent server needs an agent recovery mechanism.

The Caribbean agent server has an agent persistency mechanism that takes a snapshot of an agent whenever it modifies its data. The agent modifies its data through a callback method named "handleMessage" that returns a Boolean value. If the return value is true, the server takes a snapshot of the agent. Later, the server can recover agents from their snapshots stored in the agent log.

3.4 Grouping of Agents

Many types of agents run on the Caribbean agent server. Their behaviors can be categorized according to their roles. For example, the number of user agents created for individual users might be in the hundreds of thousands. On the other hand, the

number of service objects is typically less than 100, and none of them modifies its own data. However the objects are invoked frequently, since they are shared by all user agents. Therefore, the agent server should swap user agents in and out and keep the service objects in memory. It should also take snapshots of the user agents, but it does not need to take snapshots of the service objects.

The Caribbean agent server manages agents and service objects on the basis of groups defined by a system administrator. The administrator can set parameters limiting the number of agents in memory and the number of threads, and enable or disable the agent logging mechanism for each group.

3.5 Agent Server Cluster

A single agent server can host millions of agents because the Caribbean agent server uses the memory management mechanism described above. However, the number of agents hosted by a single agent server is actually performance limited. Therefore, clustering of agent servers is needed.

The clustering support provides a single view of a cluster for client programs and agents. A client program can send a message to an agent on any agent server in an agent server cluster. It can also multicast a message to agents on all agent servers in an agent server cluster. As well as a client program, an agent can send a message to an agent on a remote agent server through a “MessageManager” interface.

The cluster support enhances not only the performance but also the reliability of the system. The cluster recovers from a local component failure automatically by merely restarting the failed component. During the time a component is unavailable, the cluster can continue to work as long as the operations do not depend on the unavailable component. For example, creation and posting of messages to agents residing at the available servers continues even if a particular agent server is down. When the agent server recovers, the cluster recognizes it automatically, and resumes normal use of the server from that point.

4 Application Scenarios

We already applied the agent server technology to several real commercial applications. In this section, we introduce three applications. The first application is built on top of a predecessor of Caribbean and the third application is built on top of a successor of Caribbean that is compliant with J2EE. The frameworks of those agent servers are different from the framework described in this paper, but the concepts of the programming models are the same as in the programming model described in this paper.

4.1 A Commercial Service Site Providing Airline Tickets Information

TabiCan (1998-2000) was a commercial service site providing airline tickets and package tours including plane flights and hotel stays. Users accessed this server via their Web browsers and searched for airline tickets and package tours. They could obtain information from several travel agencies in a single search.

Travel agencies and users were represented as shop agents and consumer agents, respectively. The agents interacted to exchange information on airline tickets and package tours. A user instantiated his own agent on the TabiCan server and input search conditions. For example, "New York" for the destination, "Narita" for the point of departure, May 10 for the departure date, "Japan Airlines" for the airline, and "$1,000" for the maximum price. He then clicked on the "Start search" button, and his agent questioned all of the shop agents that could provide the requested airline tickets. Each shop agent queried a database several times to obtain exactly matching items and recommended items, and returned these results to the consumer agent. Recommended items were items that did not exactly match the consumer's search conditions but that the shop hoped to sell. For example, a shop agent can offer a ticket on "United Airlines" whose price is only $700. Each shop has its own selling policies. After the consumer agent has received airline tickets from all of the shop agents, the agent notifies the user of the results.

Shop agents live during the server runs. Consumer agents live for two days and are removed by the system when their lifetime is over. A user can access his consumer agent many times while it is alive. Even if he switches off his computer, his agent will still be alive, so he can access it again when he switches the computer back on.

4.2 A Notification Service for Profits and Losses for Foreign Currency Assets

The notification service is provided through a Web browser. A user registers information on foreign currency assets at the Web site. The user chooses foreign currencies and registers for the amounts in each foreign currency and sets notification thresholds for profit (loss) for each asset. Since the currency rates are updated daily, the profit (loss) of the assets changes daily. If the profit (loss) crosses the threshold set by the user, an email notification of the value is sent. For example, a user buys US$ of 1,000,000 yen at 110 yen, and also sets profit and loss thresholds of 50,000 yen. When the US dollar rate changes to 120 yen, the profit is +90,909 yen. Since this is over 50,000 yen, the value is sent by email. After the notification, the registered threshold is disabled until the user resets it. The threshold of US dollars is displayed with a mark indicating a notification threshold.

In this service, an agent is created for each user. An agent has information on its owner’s foreign currency assets and a threshold. A service object manages information on currency rates. An agent can obtain the latest currency rates from the service object by invoking a method of the service object. When rates of currencies are updated, the service object gets an update event and retrieves the latest currency rates from a database. Then, the service object multicasts an event to all agents. When an agent receives the event, it obtains the latest rates of the currencies that its owner has registered and calculates profits (losses). If profit (loss) exceeds the threshold which the owner has registered, then the agent sends a notification mail to its owner.

4.3 Location-Aware Personalized Information Notification Service

Goopas is a location-aware personalized information notification service provided by Omron. It is coupled with automated ticket gates in railway stations, and when a user of the service passes the gate, it can send e-mail messages to the user’s mobile phone.

The users input their properties such as age, job, and personal preferences when they register in the system, so the e-mail includes information that is not only associated with the station's neighborhood, but is also personalized. In this system, an agent is created for each user. The agent has its owner's properties. When a user passes through an automated ticket gate in a train station, an event indicating that the user has passed through the gate is sent to the user's agent. The agent sends mail containing relevant information about the station where the user is. The mail is sent from the system within a couple of seconds after passing through the automated ticket gate. Since the system load to create the mail for each user is significant, it is difficult to send the email quickly. Therefore, the mail content is prepared during the night or at times when new information is added to the system. An agent keeps the generated email content and sends it when the user passes through an automated ticket gate.

5 Discussion

We have applied the agent server technology to applications that handle events using personalized information. Through these projects we learned that the agent programming model described in this paper is easy for application developers to understand.

5.1 Reactive Agent Model

In the applications we developed, when a system receives an event, it executes processes for individual users. An executed process is isolated for an individual user. This means that a process for one user does not access information on other users. The process is started when a system receives an event. The model of reactive agents is suitable for these applications. The agent programming model in the agent server is based on these reactive agents [10]. However, the agent programming model adds restrictions on a thread-based model of reactive agents. A reactive agent can continue to work after it receives a message. This means that the agent can hold a thread. However, in the agent programming model, the threads are resources pooled in an agent server runtime module. An agent server gives a thread to an agent when the agent handles a message. When the agent finishes handling a message, the thread has to be returned to the thread pool and the agent server runtime module gives the thread to another agent. From the point of view of the model of reactive agents, this seems to be a strong restriction. However, the restriction was not problematic for the applications we developed.

5.2 Simple Programming Model

Since the agent programming model is simple, even non-technical people can understand the model. They can think of an agent's activities as similar to a person's activities. In the application we developed, even the non-technical people who were in charge of business processes could determine the roles of agents and define the agents appearing in the applications. Of course, the definitions were very rough and an actual application designer had to modify the definitions in the design phase, but the definitions were roughly correct. Another interesting point is that the non-technical

people who were in charge of business processes and the IT architects could talk with each other by referring to the agent model of the target application. This point is important because it can reduce problems caused by miscommunications.

5.3 Concurrent Development

The programming model also makes designing and developing an application easy. During typical design activities using the model, a designer first identifies roles appearing in the target application. Second, he or she decides on the agents to be used in the application. Third, the designer designs the messages that will be exchanged among the agents or external programs. An initial rough design is completed at this point. Then he or she starts designing each agent and each service object. This can be done separately for each type of agent because each agent is isolated in the agent programming model. There are three benefits of this approach. First is that a designer can do the implementation design in a natural way based on the initial rough design. At this stage, the designer can delegate the design of each agent to the sub-designers. Second is that the developer can understand the roles of each agent intuitively and clearly. Third is that multiple developers can work concurrently with few dependencies among agents, since each developer can develop each agent separately.

In the applications we developed, agents occasionally need to use common resources. In the notification service of the profits and losses for foreign currency assets, each agent has to convert a foreign currency code to a foreign currency name to show it on the user's Web browser. The converter component is a resource used by the agents. In the agent programming model, such components are implemented as service objects. Service objects are very useful for developing real systems. It also provides high extendability.

5.4 Development Duration

These characteristics give significant benefits for developing an application quickly. In each of the projects for the applications described above, the duration of design and development was a few months. When the time for design and development is short, the writing of code is often started before the detailed application specifications are fixed. In the agent programming model, designers and developers can have a clear view of the application structure. For example, designers and developers can quickly have a clear grasp of the area for modification caused by updating an application specification. In most cases, the modification is limited to a particular agent or a particular service object.

6 Conclusion

We introduced the Caribbean agent programming model and the framework of the model in this paper. The agent programming model is based on the "reactive agent" model. In the typical applications using the framework, each user has an agent in a server. The number of agents is very large, perhaps hundreds of thousands or more. The runtime module for the framework needs powerful capabilities to support such large numbers of agents. The Caribbean runtime provides a memory management

mechanism and a thread management mechanism. The system also needs to ensure that agents remain alive even if a system hosting the agents fails. The runtime has a recovery management mechanism. A single server might not be sufficient to host millions of agents in a system. Caribbean supports a server clustering function.

We applied the agent programming model and the framework to several commercial systems that handle events using personalized information. Through the development of these systems, we proved that the designers, developers, and even non-technical people could easily understand the agent programming model. This characteristic provides significant benefits in developing an application quickly. The applications we applied the agent programming model to are applications that handle events using personalized information. We expect that the model can offer benefits to other types of applications, such as multiagent simulations.

References

[1] Aglets System Development Kit, http://www.trl.ibm.com/aglets.
[2] G. Yamamoto and Y. Nakamura: Architecture and Performance Evaluation of a Massive Multi-agent System, Autonomous Agents '99.
[3] Y. Nakamura and G. Yamamoto: Aglets-Based e-Marketplace: Concept, Architecture, and Applications, IBM Research, Tokyo Research Laboratory, Research Report, RT0253 (1998).
[4] Y. Nakamura and G. Yamamoto: An XML Schema for Agent Interaction Protocols, IBM Research, Tokyo Research Laboratory, Research Report, RT0271 (1998).
[5] G. Yamamoto and H. Tai: Architecture of an Agent Server Capable of Hosting Tens of Thousands of Agents, Research Report RT0330, IBM Research, 1999 (a shorter version of this paper was published in Proceedings of Autonomous Agents 2000, ACM Press, 2000)
[6] G. Yamamoto and H. Tai: Performance Evaluation of An Agent Server Capable of Hosting Large Numbers of Agents, Proceedings of Autonomous Agents 2001, ACM Press, 2001
[7] H. Tai and G. Yamamoto: An Agent Server for the Next Generation of Web Applications, The 11th International Workshop on Database and Expert Systems Applications (DEXA-2000), IEEE Computer Society Press, 2000
[8] G. Yamamoto and H. Tai: Event Distribution Patterns on an Agent Server Capable of Hosting a Large Number of Agents, Research Report RT0382, IBM Research, 1999
[9] G. Yamamoto and H. Tai: Agent Server Technology for Next Generation of Web Applications, 4th International Conference on Computational Intelligence and Multimedia Applications, IEEE Computer Society Press, 2001
[10] J. M. Bradshaw, ed., Software Agents, The MIT Press, 1997
[11] K. P. Sycara, Multiagent Systems, pp. 79 - 92, AI Magazine, Summer 1998.
[12] P. A. Bernstein, and E. Newcomer, Principles of Transaction Processing, Morgan Kaufmann Publishers, Inc. 1997