October 11, 2000
RT0382
Engineering Technology   7 pages

Research Report

Event Distribution Patterns on an Agent Server Capable of Hosting a Large Number of Agents

Gaku Yamamoto and Hieki Tai

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Limited Distribution Notice
This report has been submitted for publication outside of IBM and will be probably copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the expected transfer of copyright to an outside publisher, its distribution outside IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or copies of the article legally obtained (for example, by payment of royalities).

Event Distribution Patterns on an Agent Server Capable of Hosting a Large Number of Agents

Gaku Yamamoto
IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato-shi
Kanagawa, Japan
+81-46-215-4639
yamamoto@jp.ibm.com

Hideki Tai
IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato-shi
Kanagawa, Japan
+81-46-215-5260
hidekit@jp.ibm.com

ABSTRACT

Services which handle events occurring at servers are emerging recently in the Internet world. These services will be performed in accordance with each user's preference data. To offer such services, an agent-based architecture where an agent resides at a server as a user's proxy has been proposed. In such agent systems, an event distribution pattern which sends an event to agents when a database is updated is necessary. When we build large systems hosting hundreds of thousands of agents, we have to consider system-level factors such as performance and resource limitations in addition to complexity of programming. In this paper, we describe typical event distribution patterns for large systems of this kind. Important points to consider regarding the agent interaction patterns in such systems are identified.

Keywords

Agent Design Patern, Agent Architecture, Agent Server

1. INTRODUCTION

Recently, in the Internet world, services which handle events occurring at servers are appearing. These services evolving towards personalized services which are processed in accordance with each user's preferences. To build such services, an agent-based architecture where agents reside at a server as the users' proxies has been proposed [1, 2]. In this architecture, an agent collects a user's data. A user accesses his own agent to obtain some information using a Web browser. He can also update and manage his own data by accessing his agent. The agent captures events occurring at the server and responds appropriately to the events. For example, in case of an agent monitoring foreign currency bank accounts, the agent might monitor to see when the currency rates are updated, then obtain the latest rates, calculate the user's profits and losses, and notify the user of the values by email if they exceed a threshold set by the user.

LEAVE BLANK THE LAST 3.81 cm (1.5")
OF THE LEFT COLUMN ON THE FIRST PAGE
FOR THE COPYRIGHT NOTICE

In such agent systems, an event distribution mechanism is needed to send an event to agents when a database is updated. From another perspective, the mechanism can be thought of as created by the agents' behavior patterns. In most of the research on agent behavior patterns, the patterns are studied from the viewpoint of each agent's behavior. However, in the case of a system managing a large number of agents, the situation is different from a small system. In a system managing hundreds of agents, it is reasonable that each agent obtains the latest data from a database when the database is updated. However, such behavior cannot be adapted to a system hosting millions of agents because the system might would the database millions of times. In that case, we should use a different behavior pattern that the agent host system first reads the updated data into memory from the database, and each agent obtains the data from local memory. However, to use this distribution pattern, we have to take into account the amount of updated data because of memory limitations of the system. Thus, when we build such large systems, we have to consider system constraints such as performance and memory limitations.

This paper shows three typical patterns for distributing the events that cause agents to respond. We discuss these patterns from the viewpoint of application characteristics, the number of agents, system efficiency, and complexity of programming. We also describe categories of applications for which those distribution patterns are suitable. Since these three patterns might cause system overloads and performance problems in applications where the number of agents and the number of data items are large, we describe other efficient distribution patterns applicable for the applications.

We discuss each pattern by showing the flow of messages which the pattern uses, because this reveals the system characteristics of each pattern. However describing a general framework implementing each pattern is important, so a focus of this paper is a discussion of the patterns from the system-level perspective.

In Section 2, we briefly introduce an architectural overview of an agent server hosting a large number of agents. Section 3 describes the basic structure of event distribution mechanisms which cause agents to start their processing when a database is updated. Section 4 describes some considerations of an event distribution mechanism that can be applied to a system which hosts a large number of agents. In Section 5, we divide applications into six categories from the viewpoints of both the number of agents and the number of items of data. Three common event distribution patterns are presented and discussed in Section 6. In Section 7, we describe event distribution patterns for applications where the number of agents and the number of data items are very large. Section 8 concludes the paper.

2. An Architectural Overview of an Agent Server a Hosting Large Number of Agents

The agent server described in this paper is a server that provides many users with event handling services, information monitoring services, or asynchronous processing services. Those services are provided in accordance with an individual user's preference data. For implementing those services, an agent server architecture where each user owns his agent on the server is proposed in [1, 2]. The agent is a reactive agent which starts processing in order to respond to a received message. Each agent has its own user's data, and works using that data.

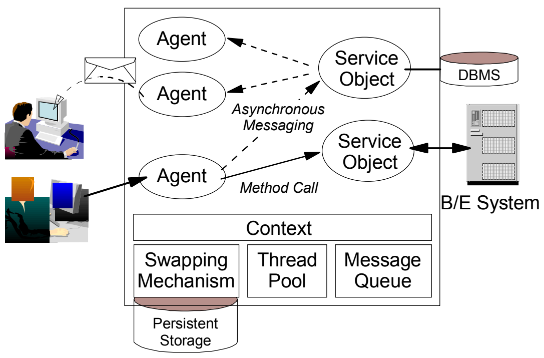


Figure 1. An Architectural Overview of the Agent Server

An architecture of such an agent server is described in [3]. The agent server is a single process running under some operating system. An agent is an object having both business logic and a user's data. All agents in the agent server share the same memory space, but an agent cannot make reference directly to other agents. It sends an asynchronous message to the other agent to communicate with that agent. The message is put into the agent's message queue. An agent server gets the message at an appropriate time and hands the message to the agent by calling the agent's message handler. The server assigns the agent a thread in a thread pool. The server limits the number of threads running on the server by using the thread pool in order to avoid system overload.

The agent server may manage hundreds of thousands of agents. In that case, the server might not be able to keep all agents in memory. Therefore, the server needs an agent swapping mechanism which swaps agents in and out between memory and storage. The mechanism converts an agent to a byte array and writes the array on the file system. At the same time, it reads another byte array from the file system and restores an agent from the read array. The process is done at the time when the server obtains a message from an agent message queue.

The agent server also has an agent recovery mechanism so that agents in memory will not be lost if a system fails. The server takes a snapshot of an agent if the agent's data has been updated after the agent processed a message.

There are also ServiceObjects which provide agents with common services such as a database access interface. The ServiceObjects have the same asynchronous messaging mechanism as agents, allowing them to send and receive messages. An agent can have object references to the service objects and can call them by method invocation.

Some of the benefits of this agent server architecture are that it is easy to develop a system because the system is modeled in a natural way [4] and it can achieve high performance if the system has a large memory space because the users' data are stored in agents located in memory [5].

3. Basic Structure of Event Distribution Patterns

When data stored in a database is updated, agents which are interested in the updated data obtain the data and start processing. On such systems, the data shared among all agents is managed by a data management object which is one of the ServiceObjects. An agent obtains data stored in the database from the data management object.

An agent has to obtain the latest data when the data are updated. How this update notification is distributed is an event distribution pattern. When the data is updated, the data management object reads the data from the database. Then it chooses agents that should start processing with the latest data, and sends events to those agents. When an agent server assigns a thread to an agent, the agent starts processing and updates the agent's data if necessary.

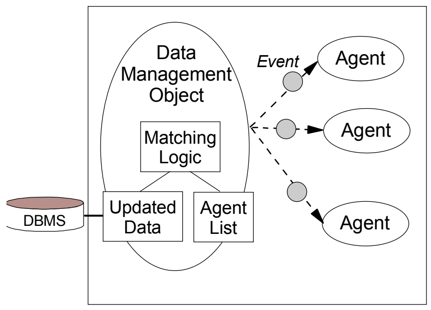


Figure 2. Basic Structure of an Event Distribution Pattern

The event distribution pattern seems not to vary at this abstract level. However, we can see several different patterns by considering deeply the processes of the data management object and of the agents.

4. Significant Points of the Event Distribution Patterns for Large Numbers of Agents

Efficiency of software programming is a focal point in most discussions about software design patterns [6, 7]. However there are few discussions of the performance of agent design patterns, except for [8, 9]. Nevertheless, in case of applications running as server software, we have to consider the efficiency of the utilization of system resources, such as CPU and memory, as well as the efficiency of the application software itself. Especially, we have to carefully consider memory utilization. Because if a system uses memory over physical memory of the computer the system will be in thrashing or the system will fail . In this section, we describe important points to consider regarding event distribution patterns on systems hosting a large number of agents.

Memory Occupied by Events

An event is sent to an agent from the data management object using the asynchronous messaging mechanism. The event is stored in the agent's message queue. Therefore, an agent server may temporarily keep many events in memory. If the size of a typical event object is a hundred bytes and the data management object creates a million event objects in order to distribute the events to a million agents, the server temporarily uses a hundred megabytes of memory, a significant amount.

Memory Occupied by Updated Data

If the amount of updated data is too large for an agent server to keep in memory, the event notifications must not themselves contain the updated data. Otherwise the agent server can fail because the events are kept in the agents' message queues and the server will try to keep all of the updated data in its memory.

Database Accesses

We also have to take account of the number of database accesses. If the data management object keeps no data in memory, it has to read the latest data from a database whenever an agent tries to obtain the data after the agent receives an event. This can cause too many database accesses. For example, if a million agents are running on an agent server, a million database accesses will be triggered for each event. It would cause system overload.

Agent Swapping

The agent server has an agent swapping mechanism as described above. Since each agent swap causes a disk access, system performance will be greatly reduced if too much agent swapping is taking place.

5. Application Categories

When we consider event distribution patterns of an agent server, we have to take into account system parameters in addition to the process flow the event distribution patterns that accompany database access, a key point is whether or not all updated data can be stored in memory. Therefore, we can divide applications into the following two types:

A: applications where all updated data can be stored in memory
B: applications where only part of the updated data can be held in memory

Another key point is the number of agents. Applications can be divided into following three types from the perspective of the number of agents:

1: the number of agents is small
2: the number of agents is large but only a minority of the agents will react to the database update
3: the number of agents is large and most of the agents will react to the database update

The meaning of "the number of agents is small" is that all the agents can respond to the updated data during a period that is short in comparison with performance demands of a particular application.

Using the above categorization, applications can be divided into six types. We will refer to each category as A-1, A-2, and so on.

6. Event Distribution Patterns

In this section, we consider three types of event distribution patterns, and discuss these patterns from the viewpoints of complexity of programming, system performance and memory utilization.

6.1 Broadcast-Event Pattern

Description

When a database is updated, a data management object distributes events to all agents. The event contains information which notifies agents of the database update. It does not include the updated data. The event can be shared among all agents. Each agent decides for itself whether or not it is interested in the specific event. If an agent needs to respond to the event, the agent obtains the updated data from the data management object by invoking a method of the data management object, for example "getData(String name)" in Figure 3. The data management object typically reads the updated data into memory prior to distributing the event if the agent server has enough memory.

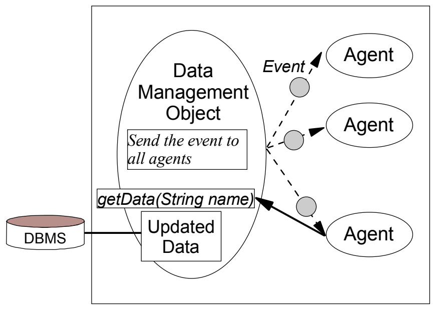


Figure 3. Broadcast-Event Pattern

Complexity of Programming

In this pattern, the data management object distributes events to all agents. Therefore, it does not need any matching logic for choosing appropriate agents. It does not need to have a mechanism for managing information on agents' interests. An application developer does not have to provide any mechanism to store the information into persistent storage. Thus it is easy and inexpensive to program for this distribution pattern.

System Performance

In this pattern, all agents will be invoked even if some agents do not need to handle the event. Therefore, if only some of the agents react to the event, this pattern is not efficient. Especially, in the case that all agents cannot be held in memory, it causes unnecessary agent swapping. On the other hand, if most agents do react to the event, this pattern is efficient because the data management object does not need to manage information on the subscriptions of agents, and it also doesn't need to worry about the choosing of agents.

This pattern can achieve very high performance if all agents and updated data are located in memory. Even if only part of the updated data can be held in memory, this pattern can work without causing system failures because of memory allocation problem, but its performance in that case is not so good.

Memory Utilization

In this pattern, all agents can share an event object. The event object does not contain any updated data. Therefore, the amount of memory required by the pattern for distribution overhead is negligible.

Types of Applications

This pattern is suitable for applications of type A-1, A-3, and B-1. It also is applicable for applications of type B-3 if the application does not require high efficiency of CPU utilization. However, in that case, the load imposed by the database management system might be heavy.

6.2 Multicast-Data Pattern

Description

In this pattern, the data management object refers to information on the subscriptions of the agents, and creates an event for each agent which subscribes to the updated data, and sends the event to the destination agent. The event includes the updated data, so the agent obtains the latest data directly from the event.

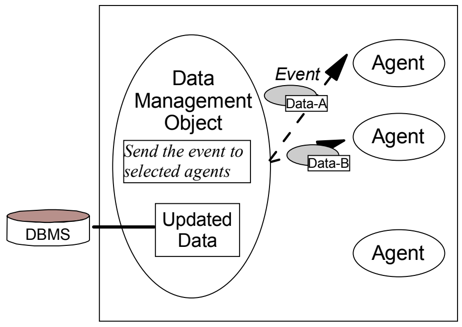


Figure 4. Multicast-Data Pattern

Complexity of Programming

The data management object has to manage information on the subscriptions and interests of the agents and chooses agents which should receive the latest data. This information must be persistent in spite of system failures. Application developers have to implement logic which manages the persistency of the information and choose appropriate agents quickly. If the implementation is not efficient, system performance will be poor.

Agents can explicitly detect items which have been updated because the received event contains all updated items. If the agent needs to start an action after receiving all updated data, this pattern is useful.

System Performance

This pattern does not send unnecessary events to agents. It is efficient in the case of systems where only some of the agents react to the event.

Memory Utilization

Each event includes the latest data which the destination agent needs. The event cannot be shared among other agents. Therefore, many events will be created temporarily. There might be hundreds of thousands of events. Moreover, all updated data may be kept into memory, possibly even in many copies. Therefore, application developers have to consider memory usage carefully.

Types of Applications

This pattern is suitable for applications of type A-2. However, it requires consideration of memory utilization.

6.3 Multicast-Event Pattern

Description

In this pattern, the data management object creates an event and sends it to agents which the data management object chooses by referring to the information on the subscriptions of agents. The event includes only information notifying the agents of the database update, but no updated data. The event can be shared among all agents. After an agent receives the event, it obtains the latest data from the data management object by invoking a method of the data management object, for example "getData(String name)" as shown in Figure 5. The data management object may read the updated data into memory prior to the distribution of the event if the agent server has enough memory.

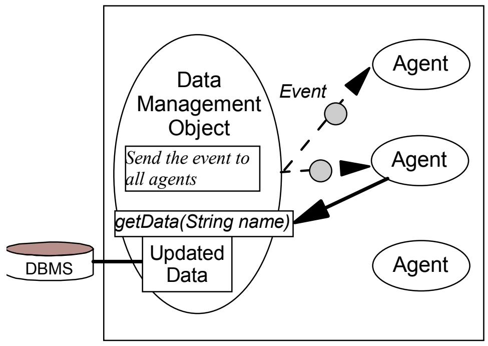


Figure 5. Multicast-Event Pattern

Complexity of Programming

The data management object has to manage information on the subscriptions of agents and choose agents which should receive the latest data. This update status information must be persistent against system failure. Application developers have to implement logic which insures the persistency of the information and chooses appropriate agents quickly. If the implementation is not efficient, system performance will be poor.

System Performance

This pattern does not send unnecessary events to agents. This is efficient in the case of systems in which only some of the agents react to the event.

Each agent obtains the latest data from the data management object by using method invocation. If all updated data can be held in memory, the operation can be done very quickly. Even if only part of the updated data is held in memory, this pattern can work without causing system failures because of running out of memory, but its performance is not so good.

Memory Utilization

In this pattern, all agents can share an event. The event does not contain the updated data. Therefore, the amount of memory required as overhead for this distribution pattern is negligible.

Types of Applications

This pattern is suitable for applications of type A-2. It also is applicable for applications of type B-2 if the application does not require high efficiency CPU utilization. But in that case, the system load from the database management system might be high.

7. Case of Many Agents and a Large Amount of Data

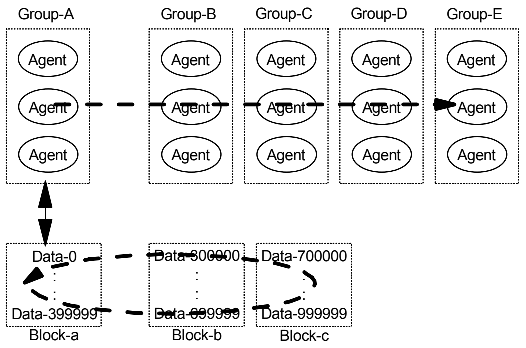
Even in the case that an agent server cannot keep all agents in memory and cannot keep all of the updated data in memory, the Broadcast-Event pattern and the Multicast-Event pattern are applicable because they don't cause system failures because of running out of memory. However, system performance is decreased because these distribution patterns may cause either many database accesses or a lot of agent swapping. As the result, these patterns may not be suitable for many target applications. In this section, we propose the basic idea of a pattern which can be applicable for such applications. Our assumptions about the applications in this section are that only some of the agents are held in memory, only part of the updated data is kept in memory, and all agents need to process the updated data.

For the situation of our assumptions, the highest costs come from either database accesses or agent swapping, and we should try to reduce these costs. Therefore, we divide the agents and the data into several agent groups and several data blocks, respectively. The sizes of those are set so that all the elements of a group/block can be held in memory at the same time. Let's consider a case where there are one million agents, and the agent server can keep two hundred thousand of agents in memory. There are one million items of data and the agent server can keep four hundred thousand the data items in memory along with the 200,000 agents. Agents are divided into five groups; "A", "B", "C", "D", and "E." Each group has two hundred thousand agents. Data items are divided into three groups; "a", "b", and "c." The group "a" contains four hundred thousand data items, and the groups "b" and "c" each contain three hundred thousand data items. The agent server loads each agent group and each data grouping into memory in turn, and performs the processing of the events handled within that combination of the loaded agents and the loaded data. We also have to schedule the order of reading the groups into memory. There are two common scheduling policies, scheduling to minimize the amount of agent swapping, or scheduling to minimize the amount of database access. We call the two approaches the "Minimum Agent Swapping Approach" and the "Minimum Database Access Approach", respectively.

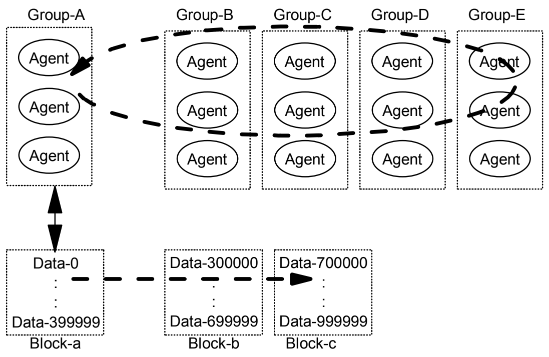
In the Minimum Agent Swapping Approach, agent groups and data groups are read into memory in order of A-a, A-b, A-c, B-c, B-b, B-a, C-a, C-b, C-c, D-c, D-b, D-a, E-a, E-b, and E-c. In this approach, each agent group is fixed and held in memory until all agents of the group finish their actions. Moreover, we can reduce the number of times of database accesses by reversing the order of reading the data groupings each time an agent group is finished--this reuses the last data group so it does not need to be reloaded. In the Minimum Data Access Approach, the agent groups and data groupings are read into memory in the order of a-A, a-B, a-C, a-D, a-E, b-E, b-A, b-B, b-C, b-D, c-D, c-E, c-A, c-B, and c-C. Each data group is fixed until all items of data of the grouping have been handled completely. Again, we can reduce the amount of agent swapping by reversing the order of reading the agent groups between each pass.

Let's assume that in the initial state agent group A is in memory. In the Minimum Agent Swapping Approach, the amount of agent swapping and the number of data accesses are 800,000 and 3,400,000 respectively. In the Minimum Data Access Approach, the amount of agent swapping and the number of data accesses are 2,400,000 and 1,000,000 respectively. We cannot say which approach is definitively better because it depends on the performance of the agent swapping routines and the database access routines. From the view of the amount of data read from disk, an agent is probably bigger than an item of data, because an agent has several components. On the other hand, database access may take longer than agent swapping, because it causes inter-process communication, data format conversions, involves copying the data several times and so on. Further research on the relative performance of agent swapping and database access is required.

A purely restricted asynchronous messaging mechanism is not adequate for implementing this approach. To exchange an agent group with another agent group, the data management object needs to wait until all agents of the current agent group have completed their event handling. But an asynchronous messaging mechanism normally do not have a mechanism for waiting for completion, so the data management object cannot detect the event. We need such a waiting mechanism, for instance, a mechanism which calls a method of the data management object when all agents of the current agent group have completed their event handling. Unfortunately, our agent server does not have such a mechanism, though we are planning to implement it.



(a) Minimum Agent Swapping Approach



(b) Minimum Data Access Approach

Figure 6. Event Distribution Patterns using Groups

8. Conclusion

In this paper, we described considerations of agent patterns on an agent server which hosts a large number of agents. In such agent servers, system metrics such as CPU utilization and memory must be considered as well as complexity of programming. We described three event distribution patterns; Broadcast-Event, Multicast-Data, and Multicast-Event. Characteristics of each of these patterns were also discussed from the viewpoints of complexity of programming and system considerations. Those three patterns have problems under the conditions where all agents and updated data cannot be loaded into memory at the same time. We proposed a base approach for an event distribution pattern under thees conditions. The approach of grouping agents and data is efficient because the costs of agent swapping and database access can be reduced. However, it requires an extra mechanism for synchronization. Unfortunately, our current agent server does not have such a mechanism, though the considerations of this paper require us to add it if our server will be required to handle such large-scale applications.

9. REFERENCES

[1] IBM Caribbean, <http://www.alphaworks.ibm.com/tech/caribbean>

[2] Kinetoscope VIA Systems, <http://www.kinetoscope.com/via>

[3] G. Yamamoto and H. Tai: "Architecture of an Agent Server Capable of Hosting Tens of Thousands of Agents", IBM Research Research Report RT0330, 1999 (a shorter version of this paper was published in Proceedings of Autonomous Agents 2000, ACM Press, 2000)

[4] 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

[5] G. Yamamoto and H. Tai: "Can a Persistent Object Technology Make A High Performance Application Server?" , WIT-2000, Sep. 2000 (in Japanese)

[6] E. Gamma, R. Helm, R. Johnson, and J. Vissides: Design Patterns, Addison-Wesley

[7] Y. Aridor and D.B. Lange: "Agents Design Patterns: Elements of Agent Application Design" , Proceedings of Autonomous Agents '98, ACM Press, 1998

[8] O.F. Rana and K. Stout: "What is Scalability in Multi-Agent Systems?" , Proceedings of Autonomous Agents 2000, ACMPress, 2000

[9] O.F. Rana: "Performance Management of Mobile Agent System" : Proceedings of Autonomous Agents 2000, ACMPress, 2000