View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by AIS Electronic Library (AISeL)

Association for Information Systems
AIS Electronic Library (AISeL)

AMCIS 1999 Proceedings

Americas Conference on Information Systems (AMCIS)

December 1999

An Agent-Based Approach For Collaborative Schema Design

Benjamin Khoo
University of Maryland

Sriram Chandramouli
Hughes Network Systems, Maryland

Follow this and additional works at: http://aisel.aisnet.org/amcis1999

Recommended Citation

Khoo, Benjamin and Chandramouli, Sriram, "An Agent-Based Approach For Collaborative Schema Design" (1999). *AMCIS 1999 Proceedings*. 19.
http://aisel.aisnet.org/amcis1999/19

This material is brought to you by the Americas Conference on Information Systems (AMCIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in AMCIS 1999 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.

An Agent-Based Approach For Collaborative Schema Design

Benjamin Kok Swee Khoo (khoo@gl.umbc.edu)
Department of Information Systems
University of Maryland, Baltimore County

Sriram Chandramouli (schandramouli@hns.com)
Hughes Network Systems
Germantown, Maryland.

Introduction

Data modeling is a crucial step towards incorporating successful databases in an organization. The design, operational behavior, and use of a database are affected by the meaning of the information it manages. The cognitive capability of the human mind is rather complex; it has the capability to visualize a problem domain in a variety of perspectives. It would be interesting to conceive of an application, by which the best thoughts of human minds can be pooled together to create a conceptual schema design for a database. Thus, the database design would be enhanced if capabilities were provided for collaboration between *different designers* working on *different platforms* at *different locations* and even at *different times* but working on the *same database design*. This paper describes an agent-based approach, using an agent-based architecture communicating through the Internet, which promotes the collaborative conceptual schema design. The agent communication was to be developed using KQML (Knowledge Query Manipulation Language) in Java. The prototype of the system developed captures only the static properties of a system. The dynamic aspects of the operations are resolved by considering certain additional aspects that are not exactly database objects, but are associated with a database occurrence which changes as a result of an operation. These are implementation issues that must be taken care by a database designer during the implementation.

System Overview

The intent of the collaborative schema design project is to enable different database designers working on different platforms at different locations, and even at different times to work on the same database design. This feature is achieved through an agent communication paradigm that allows agents to broadcast their available services to prospective consumers. The clients are provided with a GUI stub that enables them to type in text commands for creating entities and the relationships between them. The longer term plan is to provide the clients with a front-end Java enhanced graphical tool (to replace the GUI stub) that consists of a workspace and a palette that can be used for creating various entities, and the relationships between them.

Every client expresses his interest to messages broadcasted by registering with the same group identity as the other design team members or clients. Messages can be shared between clients, only, when the clients share the same group identity. When a client opens a socket connection and registers with the MultiServer, the MultiServer creates a client identity and an agent that speaks Knowledge Query Manipulation Language (KQML) is spawned off for each client (for multi-clients, multi-threads of agents will be spawned). This agent will act as a unique representative for that particular client. The communication henceforth will take place between the different agents that act as different client representatives. Each agent spools its client request (typically SQL statements) to the Agent Server. The Agent Server invokes the SQL statements on the Postgres95 database, thereby storing the entities, relationships and attributes created or changed. On completion of the request, the Agent Server will return the words "Success" or "Failure" to the originating agent. The originating agent then forwards the packet to the other agents, which will in turn send it to their respective clients. In this way, any changes to the database design by any client is "broadcast" to all other clients working on the same database design. There will be some basic rules to control modification of the database design.

These database integrity checks are enforced by the agent server (for example, clients cannot create entities that already exist, and relationships cannot be defined between the same set of entities, if a relationship already exists between them, etc). Since the collaborative schema design allows entities and relationships to be shared across multiple clients, only the client that creates an entity or a relationship has the rights to modify or delete them. In this way, the notion of ownership has been enforced on the object created by the clients. The software also ensures that entities are not deleted before deleting the relationships between them. It also prevents entities to be deleted before deleting the attributes that may be contained within the entity.

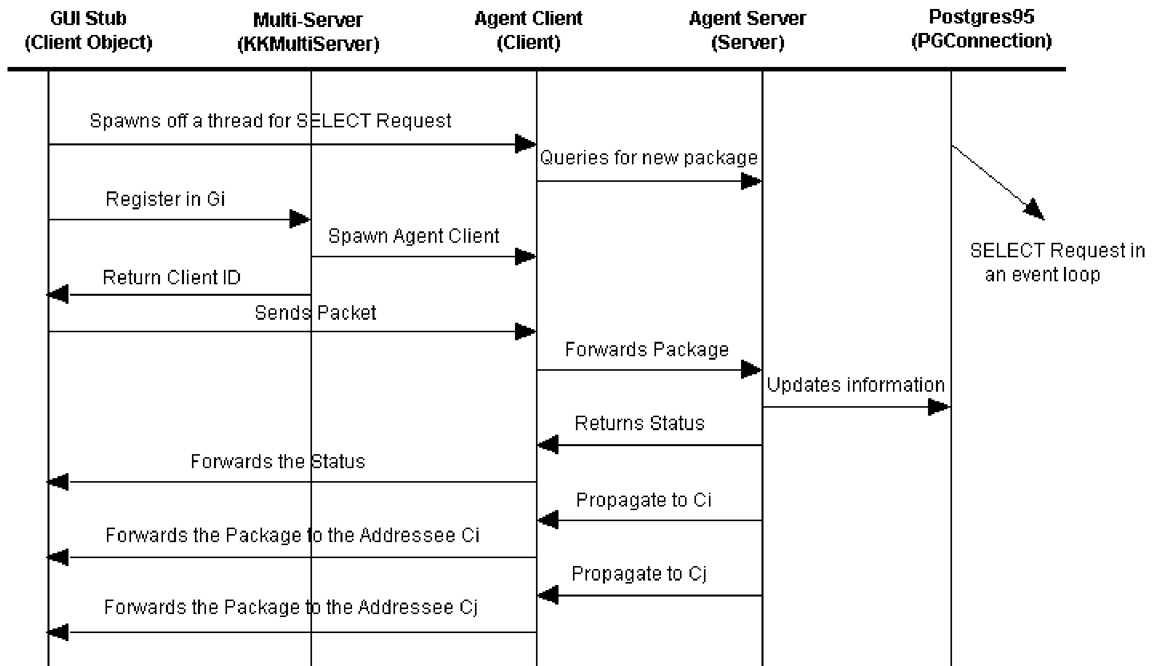
An agent server has an up-to-date information regarding the unified schema different users attempt to model. It is imperative that the agent server enforces the consistency checks between the different objects (Entities or Relationships) a client creates. When the agent representatives share information with the agent server, the agent server caches connection parameters from the agent representatives. Thus, an agent server also acts as an agent name server, as it transparently locates the agent representative to which a packet must be routed. In this way, the propagation of the packets is opaque to the clients participating in the unified schema design.

An interesting scenario arises when a client goes temporarily out-of-service or de-registers himself from the agent server. The client will no longer receive any packet broadcasted by the agent server. But, when the client process starts up again, an agent that acts as a unique representative to the client is spawned off, where it queries the agent server frequently to extract the up-to-date information. The details of such a mechanism are discussed in a greater detail in the subsequent sections.

System Implementation

One of the important issues we faced while implementing this project is how does the client obtain the information that is propagated by other clients. We considered two possible alternatives for designing this feature. The first alternative is to make the agent server proactive and broadcast a packet to the prospective clients when it receives it. The second alternative is to make the client proactive, by frequently querying the agent server for new packets. The latter alternative is chosen due to the argument that follows: an agent server is already a busy process, as it involves sequencing or synchronizing the packets generated by different clients. The agent server also performs database integrity checks and stores the information contained within a packet in a persistent store. Hence, if the application logic that involves the propagation of packets to prospective clients were given to the agent server, it would be even busier! This would seriously affect the response time to the clients, as the agent server will be threaded to perform a variety of activities. On the other hand, the client process does no more than accepting the user input, converting them into packets and sending them to the agent servers. As it stands, the client process is light in nature. Thus, the application logic involved in proactively querying for more information contained within an agent server can be incorporated on the client side.

This functionality is achieved in the following way: every client issues a select request to the agent server for querying new packets generated by other clients. Such requests can be issued in a separate thread within the client process. The requests are issued once in every time period, t. This involves running an event loop within a separate thread, that queries for new packets. Since every client process runs a separate thread that issues a select request, multiple threads from multiple clients have to be synchronized to prevent inconsistency in the resources they try to access. The thread-locking feature in JAVA that allows event synchronization created by different threads is utilized for this purpose. This is illustrated in the following event trace diagram.



Looking at the low-level design of the project, there are various packets generated by the clients. These packets are transmitted to the agent server through an agent spawned off by every client process (through the MultiServer), that acts as a unique representative of the client. We identified nine different packets that capture the entire functionality needed for the conceptual schema design.

Work Completed and Future Scope

The entire communication aspect of the project that promotes group schema design has been implemented. The software tools that were used are the JAVA Development Kit 1.1, JACKAL, a JAVA API for creating agents that speak KQML, and a JAVA API for interfacing with the Postgres95 database. Currently, GUI stubs act as substitutes to a fully functional GUI interface, which is to be developed in the future. The GUI front end is going to be a graphical module that provides clients with a workspace for modeling the conceptual schema design for the database. Once the properties of the objects (Entities/Relationships) have been defined for the object, the packets, as described before, can be generated and streamed across to the agent server through the different agent representatives. The GUI module also runs a separate thread that issues select requests frequently to query for packets generated by other clients. Once the GUI front end receives a packet, it can translate a packet to a graphical object and displays them on the client screen.

References

References available upon request from the first author.