

A Data Management System for Replicated Application

Guangxin (Gavin) Yang
Bell-Labs Research, Lucent Technologies
600 Mountain Ave., Murray Hill, NJ 07974 USA
gxyang@acm.org

ABSTRACT

Existing groupware toolkits are always built atop a programming language or system with some enhancements or extensions to support the development of a certain category of cooperative applications. This approach restricts the areas where the toolkits could be used. We argue that an ideal toolkit should be adaptive to different cooperation modes, a wide range of applications, and different programming languages. We present a groupware toolkit for replicated applications that meets these requirements by building it from a data management point of view. The architecture and certain important runtime supports are briefly explained.

Categories and Subject Descriptors

H.2.4 [Information Systems]: Database Management - *systems*

General Terms: Management

Keywords: Groupware, CSCW, Data Management, Cova

1. INTRODUCTION

We describe a novel data management approach based system for developing replicated cooperative applications. Our goal is to provide a toolkit adaptive to multiple cooperation modes, a wide range of application areas, and different programming languages. This is achieved by:

- An object-oriented (OO) data model and programming language for application adaptability, i.e. being useful in a wide range of application domains, e.g. *virtual environment*, *form processing*, *design*, and *document processing*, to name just a few.
- Novel optimistic concurrency control and implicit session management mechanisms for cooperation adaptability, i.e. being able to support multiple cooperation modes, e.g. *synchronous*, *autonomous*, *asynchronous*, within the same framework.
- A data management based approach for language adaptability, i.e. being able to work with a wide range of languages.

2. COVA FRAMEWORK

Cova is the name of a meta-model, programming language, and runtime system designed for developing integrated *cooperative applications* [3]. By *integrated*, we mean a cooperation process where both *single user*, *synchronous*, *autonomous*, *asynchronous*, and even other *integrated activities* are involved. In this paper, we focus on how data objects are managed for supporting synchronous and autonomous cooperation.

We adopt a hybrid architecture, i.e. a centralized Cova server with multiple allied Cova clients embedded in applications running at different sites. Figure 1 depicts a runtime scenario. A Cova server maintains ❶ a collection of *class definitions*, which defines the semantic of shared objects and are compiled from programs written in the Cova object description language, an OO language based on a revised version of ODMG's object model [1]; ❷ an *object space* that contains the persistent states of shared object; and ❸ a *session manager* (SM) records how shared objects are being accessed and provides necessary controls at different stages.

Currently implemented on Win32 platforms as a dynamic link library (DLL) and linked into end-user applications, a Cova client holds the replicas of Cova objects used by applications running at a particular site and provides the programming interfaces for *manipulating objects*, *retrieving object state and awareness information*, *receiving notification about state changes*, etc.

A Cova client consists of a *request queue*, a *request scheduler*, and an *interpreter*. An application converts user interface operations to method calls on the underlying Cova objects and puts them into the request queue. The scheduler chooses according to a specific criterion a request that could be executed and executes it with the interpreter.

3. SUPPORTING REPLICATED APPS

We will now have a closer yet very brief look at how some important parts of the system are implemented.

3.1 Implicit Session Management

An object, say o , in the object space of a Cova server may be opened for accessing by one or more applications. The session object for o , denoted as S_o , is created to record these accesses. An element of S_o , denoted as s , represents an application working on o and can be formalized as:

$\langle sid, appid, state, start_time, end_time, user_name, app_name \rangle$

where sid uniquely identifies S_o within a Cova server; $appid$ uniquely identifies the application within the session; $state$ indicates how o is being accessed by the application, e.g. *connected* or *disconnected*, *shared* or *exclusive*. Other fields are self-evident.

Our scheme is *implicit* in the sense that a session object is created and updated automatically to reflect some important events. For example, S_o is created when o is opened by the first application; a new element will be added when another application opens o ; the end_time field will be set when an application exits; S_o is destroyed when the last application exists.



The implicit scheme is in contrast to an explicit one, which requires certain session management commands, e.g. *create session*, *join*, *leave*, *invite* [2]. Since an implicit scheme removes the tasks in writing codes for session management, from a developer's point of view, its advantages are obvious.

Copyright is held by the author/owner(s).

CVE'02, September 30–October 2, 2002, Bonn, Germany.

ACM 1-58113-489-4/02/0009.

3.2 Optimistic Concurrency Control

An object, e.g.   in Figure 1, may be opened and accessed by multiple sites simultaneously. A method call generated by one site is executed both locally and remotely at other sites in the same session. Concurrency control is vital to maintain the consistency among the replicas of an object. Our consistency model requires that ❶ causal dependency among method calls is respected; ❷ a method call produces the same overall effect at all sites; ❸ the final states of the replicas are identical at all sites.

The basic ideas of our concurrency control algorithm include: ❶ a state vector based approach is used to ensure causal dependency; ❷ a method call will be interpreted by the Cova interpreter; accesses to an attribute or member of an object are done through one of the four primitives, i.e. *retrieve*, *update*, *insert* and *delete*; ❸ for each component object of the shared object, an operation log will be maintained recording all modificatory primitives; ❹ each primitive is executed based on the operation log and the current state of the data item it refers so that it achieves the same effect as it does at the site where the method call was generated. The scheme is fully optimistic in the sense that no centralized sequencer is required. A full description of this scheme can be found in [4].

3.3 Smooth Joining of New Comers

A new comer to S_o needs to get o 's latest state to start with. When o is being worked on synchronously at multiple sites, the server may not necessarily hold its latest state due to unsaved modifications. SM coordinates the joining by following a protocol that includes the following steps.

❶ SM multicast to all sites in S_o a message indicating them to stop sending out local method calls; ❷ upon receiving the message, a Cova client stops accepting local method calls and sends its state vector to the SM after it executes a request; ❸ when the state vectors of all sites in S_o are identical, SM randomly requests to a client site the current state of o , adds a new element to S_o , and returns o to the new comer; and ❹ SM multicasts to all sites in S_o a message indicating the joining has been completed and they should make necessary reconfigurations, e.g. resetting the state vector, clearing operation logs, to continue the session with a new member.

When the protocol reaches step 3, all existing sites have executed all method calls generated before the joining request. According to our concurrency control algorithm, the state of o at all sites should be identical. Therefore, at step 4, all sites in S_o , including the new comer, hold an identical copy of o .

3.4 Disconnecting and Reconnecting

An application may want to disconnect from the server, keep working on o separately for a period, and reconnect to the server to commit its changes. Supporting this feature may be a key to mobile users. Some problems e.g. *merging*, *conflict resolution*, may arise when multiple applications work on o in this way simultaneously.

We use an approach that is based on our concurrency control mechanism. Just imagine the disconnection as an extremely long network delay, which may last several days. A disconnected Cova client can still execute immediately local method calls. However, these calls will not be sent out until the client reconnects.

The SM buffers all these method calls and executes them according to the concurrency control algorithm when it is sure that all

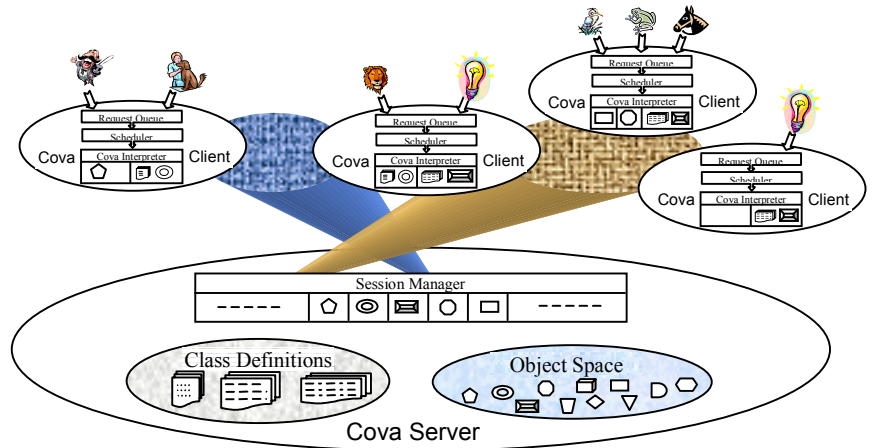


Figure 1. A runtime scenario of the Cova system.

disconnected clients have reconnected. At this time, all changes to o are merged together, o is made persistent, and the session is considered to be finished. A new request to open o will lead to the creation of a new session object and be returned the new state of o . This way, an application has the freedom to switch between synchronous and autonomous modes.

3.5 Programming Interface

An application needs to interact with the embedded Cova client for several purposes. We define a thin programming interface with which an application could:

- *Logon* to and *Logoff* from a Cova server
- *Open* an object from the server in a specific mode. An opened object could be *Saved*. An application could call *Submit* to indicate it finishes working on an object. *Execute* a method of the object with supplied parameters. Return values are copied to a caller supplied buffer.
- *Set* the name of the event that will be triggered when the object is updated so that the application could be notified for certain purposes, e.g. view refreshing.
- *Get* related awareness information. Currently we provide information about the applications and users that are working on the same object. The information is stored in the session object.

4. CONCLUSIONS AND FUTURE WORK

We have developed a data management system for developing replicated multi-user applications. The system outperforms existing systems in its adaptability to different cooperation modes, a wide range of application areas, and different programming languages.

Future development includes developing applications with graphics user interfaces based on the system. We are also planning to port the whole system to non-Win32 platforms, like Unix, Linux, and make all implementations interoperate with each other.

5. REFERENCES

1. Cattell, R. G. G., Barry, D., Bartels, D., et al. Object Database Standard: ODMG 3.0. San Mateo: Morgan Kaufmann Publishers, 2000
2. Edwards, W. K. Session management for collaborative applications. In: *Proc of ACM CSCW*, Chapel Hill, 1994, 323-330
3. Yang, G., Shi, M. Cova: a programming language for cooperative applications. *Science in China*, Series F, 2001, 44(1):73-80
4. Yang, G. Optimistic concurrency control for replicated objects. *Bell-Labs Technical Memo*, 2002