Distributed Data Management Support for Collaborative Computing*

S. P. Olesen**, S. E. Chodrow, M. Grigni, and V. S. Sunderam

Department of Mathematics and Computer Science
Emory University, Atlanta GA 30322, USA

Abstract. This paper proposes a framework for distributed data management in collaborative computing systems. The Collaborative Computing Data Space (CCDS) is a user-level tool that provides a group of participants with a space of shared data objects, as well as a simple interface to operations on data objects in the data space.

* Research supported in part by NSF grant ASC-9527186

** Presenting author. Email: olesen@mathcs.emory.edu, voice: +1 404 727 7580, fax: +1 404 727 5611

1 Introduction

Collaborative computing systems provide multiple users with facilities to cooperate in distributed and heterogeneous computing environments. Typical services provided by collaborative computing systems include common displays, multimedia conferencing applications, and (quasi-)transparent access to remote devices and computational resources. Underlying this shared environment is a data management subsystem that supports the reliable sharing of distributed data objects among collaborators.

Collaborative computing assumes a geographically dispersed group of participants (scientists, managers, students, *etc.*), working together on a particular problem. The data management support for collaborative computing should provide a group with a unified interface to shared data objects, even when those objects are of different types and from different sources. Additionally, a collaborative data management system should provide concurrent access to data objects, an authentication mechanism, consistency control, and support for heterogeneous architectures and environments. Our model of collaborative computing is inherently distributed; hence, the underlying data management is also distributed.

Existing parallel file systems provide parallel applications with access to storage in distributed heterogeneous environments. Kotz and Nieuwejaar [1] give an overview of recent parallel file systems. Most existing parallel file systems, for example PIOUS [2, 3] or Galley/Galley2 [1], are developed with parallel scientific and high-performance applications in mind. For example, the PIOUS file system provides parallel PVM applications with access to a segmented parallel file object as well as consistency and fault tolerance, implemented on top of existing filesystems. The Galley/Galley2 systems provide a core filesystem for parallel file objects in a global name space that serves higher level application programming interfaces.

In this paper, we propose a data management system supporting loosely collaborating users, rather than highly coordinated parallel algorithms. A collaborative data environment demands many of the same services of a traditional file system: naming, directories, ownership, access control, and reliability. Distributed computing requires further support to provide consistency, atomic operations, and performance.

However, as a further constraint we also want a system that is widely usable. In particular we will not assume that collaborators on Unix systems have root access, as would be necessary to install a traditional filesystem. Local filesystems will be used to maintain local directories and copies of shared objects, but in an unprivileged way.

2 The Collaborative Computing Data Space

We propose a framework based on a Collaborative Computing Data Space, referred to as CCDS. The CCDS is comprised of objects of various types that may be accessed by a group of participants in a collaborative computation. Note that both humans and applications may be participants in a collaborative computation. Participation is not necessarily associated with human interaction. We refer to objects in the CCDS as virtual objects.

2.1 Virtual Objects

Objects in the CCDS are classified by type. CCDS supports *file, directory, pipe, active,* and *device* object types. A file holds a sequence of bytes of definite length. Semantics of concurrent file access will be described in section 2.3. A directory object is a file containing a list of objects that is updated atomically. A pipe provides access to a bidirectional stream of bytes, of indefinite length; a pipe might provide access to a remote program, and could also involve multiple readers and writers. Writes to a pipe are non-blocking and writes from multiple writers are updated atomically. An active object is an executable in some portable format, such as Java bytecode that has access to an API for CCDS. A device object provides an interface to a physical device; a device might be a scientific instrument such as a microscope where one participant writes to the microscope (ie. controlling its operation), and with multiple participants reading and viewing the results. A device object is created by an active object which serves as an intermediary between the actual device and our API.

2.2 Creating Virtual Objects

A virtual object may be created in the CCDS in two ways: (1) a participant may import a data object from his/her own file system into the CCDS, or (2) a program may create a native virtual object using the CCDS programming interface. An imported object is typically an object that is created in an underlying file system and then imported into the CCDS to be shared with other participants in a collaborative computation. For example, consider a data file created by one collaborator which is to be viewed by the entire group. This file is imported into the CCDS and then may be simultaneously viewed by the group with the appropriate application. A data object that has been created with the import function may also be "unimported" back to the originating file system, thus removing it from the CCDS. An export function allows a participant to make a local copy of a virtual object without removing it from the CCDS.

2.3 Object Attributes

An important feature of the CCDS is that participants may access objects simultaneously. The semantics of concurrent access are determined by object attributes described below.

Every virtual object created in the CCDS is associated with the following set of attributes: `name`, `owner`, `token`, `size`, `timestamp`, `permissions`, `object type`, and `coherence`. The attributes must be specified when an object is created or imported; however, the `name, owner, token, permissions` and `coherence` attributes may be modified after creation by the `owner` of the object. Where an attribute only applies to certain object types it will be noted. The attributes have the following semantics.

`name:` The unique name by which the virtual object is referred to within the CCDS.

`owner:` A unique identification for the participant who controls the attributes of the virtual object; initially the participant that created/imported the object into CCDS.

`type:` Whether the virtual object is a `file, directory, pipe, active,` or `device` object.

`origin:` Whether the object is native—created within the CCDS—or imported from outside the CCDS (and if so, from where).

`token:` For objects accessed with token control, this identifies the participant who currently holds the right to modify the object. If the object does not have token control, this is nil (and anyone may attempt to modify the object).

`size:` The size of the virtual object in bytes. This is meaningless for pipes and device objects.

`timestamp:` The time that the object was last modified, according to the owner's clock.

`permissions:` Read, write, and execute permissions for owner and other participants, similar to UNIX access specifiers.

`coherence:` The coherence attribute determines the semantics of file object change updates. If the file has `real time` semantics, then every participant is guaranteed to have the same view of a file at all times (this requires locking for every operation). With `on demand` semantics, a participant may request that other participants are notified of changes to a file. Under `interval` semantics, participants are notified of changes to a file at timed intervals. Finally, under `never` semantics, participants are never notified of file changes.

`data:` Data content of object (for file and active objects).

2.4 Object Operations

An important feature of active objects is that they get access to an API for object operations in the CCDS. The operations that that apply to file objects are similar to ordinary file operations like create, open, close, read, write, and seek; furthermore, a file object may be accessed with token control using lock and unlock operations. Except for seek, the same set of operations apply to pipe and device object types. The create operation for a device is different from that for file and pipe types; with a device create operation a user has to supply active objects that implement a protocol between participants and the physical device. Finally the API includes operations for import, unimport, and export of virtual objects.

3 Implementation

We require that the CCDS is implemented as a user-level tool that may be installed and used without system root privileges on a wide variety of operating systems. This requirement makes the CCDS easy to install and use; however, it also sacrifices some flexibility and performance that could be obtained with a set of customized core (kernel) file operations (cf. [1]).

Users may access the CCDS via a CCDS user interface tool. Initially the user interface tool authenticates a participant and grants access to operate on virtual objects in the CCDS. The operations that a participant may perform include import, unimport, and export of virtual objects, change object properties, and list content of directories.

The lower level methods used for implementation of the CCDS has three main components: (1) a local filesystem at each participant that is used for storage of the data space objects; (2) a group communication package, such as Collaborative Computing Transport Layer (CCTL) [4], that provides a group of participants with reliable, atomic multicast channels; and (3) an application programming interface that provides new applications with seamless interface to CCDS objects. Updating changes to object data is a performance bottleneck in the CCDS system. Certain common cases should be optimized: for example compression of object updates, and looking for updates which are only minimally different from the previous data.

4 Conclusion

This paper describes our plans for a distributed data management support system for collaborative computing. We are currently working on defining a group communication based architecture and API for an implementation of a Collaborative Computing Data Space that may become part of a collaborating computing system .

References

1. Kotz, D., Nieuwejaar N.: Flexibility and Performance of Parallel File Systems. ACM Operating Systems Review. **2** (1996) 63–73
2. Moyer S. A., Sunderam V. S.: A Parallel I/O System for High-Performance Distributed Computing. Department of Math and Computer Science, Emory University. Technical Report CSTR-940101 (1994)
3. Moyer S. A., Sunderam V. S.: Characterizing Concurrency Control Performance for the PIOUS Parallel File System. Department of Math and Computer Science, Emory University. Technical Report CSTR-950601 (1995)
4. Rhee I., Cheung S. Y., Hutto P. W., Sunderam V. S.: Group Communication Support for Distributed Multimedia and CSCW Systems. Submitted to 17th IEEE International Conference on Distributed Computing Systems, Baltimore, MD (1997)

This article was processed using the LaTeX macro package with LLNCS style