IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Distributed Shared Virtual File System Explained using IVY System

Shilpa D. Ghode
Dept. of Computer Technology KITS, Ramtek, Nagpur, India
shil_ghode@yahoo.co.in

Abstract

The shared virtual memory provides a virtual address space that is shared among all processors in a loosely coupled distributed-memory multiprocessor system. Application programs can use the shared virtual memory just as they do a traditional virtual memory, except, of course, that processes can run on different processors in parallel. The shared virtual memory not only “pages” data between physical memories and disks, as in a conventional virtual memory system, but it also “pages” data between the physical memories of the individual processors.
The main difficulty in building a shared virtual memory is solving the memory coherence problem A prototype system called IVY has been implemented on a local area network of Apollo workstations. The experimental results of nontrivial parallel programs run on the prototype show the viability of a shared virtual memory. The success of this implementation suggests an operating mode for such architectures in which parallel programs can exploit the total processing power and memory capabilities in a far more unified way than the traditional “messagepassing” approach.

Introduction

The benefits of a virtual memory go without saying; almost every high performance sequential computer in existence today has one. In fact, it is hard to believe that loosely coupled multiprocessors would not also benefit

from virtual memory. One can easily imagine how virtual memory would be incorporated into a shared-memory parallel machine because the memory hierarchy need not be much different from that of a sequential machine. On a multiprocessor in which the physical memory is distributed, however, the implementation is not obvious. Two kinds of multiple CPU systems exist[11]: multiprocessors and multicomputers. A multiprocessor is a machine with multiple CPUs that share a single common virtual address space. All CPUs can read and write every location in this address space. Multiprocessors can be programmed using well-established techniques, but they are difficult and expensive to build. For this reason, many multiple CPU systems are simply a collection of independent CPU-memory pairs, connected by a communication network. Machines of this type that do not share primary memory are called multicomputers. The usual approach to programming a multicomputer is message passing[11]. The operating system provides primitives SEND and RECEIVE in one form or another, and programmers can use these for interprocess communication. This makes I/O the central paradigm for multicomputer software, something that is unfamiliar and unnatural for many programmers. An alternative approach is to simulate shared memory on multicomputers. One of the pioneering efforts in this direction was the work of Li and Hudak [1]. In their system, Ivy, a collection of workstations on a local area network shared a single, paged, virtual address space. The pages are distributed among the workstations. When a CPU references a page that is not present locally, it gets a page fault. The page fault handler then determines which CPU has the needed page and sends it a request. The CPU replies by sending the page. Although various optimizations are possible, the performance of these systems is often inadequate.

Ivy is a multi-user read/write peer-to-peer file system. Ivy has no centralized or dedicated components, and it provides useful integrity properties without requiring users to fully trust either the underlying peer-to-peer storage system or the other users of the file system.

An Ivy file system consists solely of a set of logs, one log per participant. Ivy stores its logs in the DHash distributed hash table[9]. Each participant finds data by consulting all logs, but performs modifications by appending only to its own log. This arrangement allows Ivy to maintain meta-data consistency without locking. Ivy users can choose which other logs to trust, an appropriate arrangement in a semi-open peer-to-peer system.

Ivy presents applications with a conventional file system interface. When the underlying network is fully connected, Ivy provides NFS-like semantics[18], such as close-to-open consistency. Ivy detects conflicting modifications made during a partition, and provides relevant version information to application-specific conflict revolvers.

Memory Coherence Problem

A memory is coherent if the value returned by a read operation is always the same as the value written by the most recent write operation to the same address. An architecture with one memory access path should have no coherence problem. A single access path, however, may not satisfy today's demand for high performance. The memory coherence problem was first encountered when caches appeared in uniprocessors and has become more complicated with the introduction of "multicaches" for shared memories on multiprocessors The memory coherence problem in a shared virtual memory system differs, however, from that in multicache systems. A multicache multiprocessor usually has a number of processors sharing a physical memory through their private caches. Since the size of a cache is relatively small and the bus connecting it to the shared memory is relatively fast, a sophisticated coherence protocol is usually implemented in the multicache hardware such that the time delay of conflicting writes to a memory location is small. Shark[19] introduces a novel cooperative-caching mechanism, in which mutually-distrustful clients can exploit each others' file caches to reduce load on an origin file server. On the other hand, a shared virtual memory on a loosely coupled multiprocessor has no physically shared memory, and the communication cost between processors is nontrivial. Thus conflicts are not likely to be solved with negligible delay, and they resemble much more a "page fault" in a traditional virtual memory system. The work on updates and transactions in peer-to-peer systems can be classified based on who is allowed to modify it, and how conflicting modifications are resolved [21]. This can be divided into the following categories:

Single owner/primary copy is a setting in which each data item that originates from some source peer p can only be modified by (or through) p — i.e., no other peers are allowed to directly modify that data.

Owner-resolver protocols [21] allow multiple peers to modify the data, and they typically rely on the owner to resolve any conflicts. If resolution is impossible, they "branch" the data into fully independent instances. Consensus protocols allow multiple peers to modify the data, and some set of nodes works together to determine how to arbitrate for consistency. Partial divergence schemes handle conflicts in a way that results in multiple divergent copies of the data, but they operate at a finer level of granularity than divergent replica protocols, and they allow some portions of the data instance to remain shared even after "branching" two instances. In the simplest schemes, each data item is owned by a single source, which may update that data. Many other nodes may replicate the data but may not change it (except, perhaps, by going through the primary copy at the owner). This is sometimes referred to as the single-writer, multiple readers problem. In this type of scheme, the owner of the data uses a timestamp (logical or physical) to preserve the serial order of updates, or to arbitrate among different verions of the data. Since there is a single owner and a single clock, any node can look at the data and deterministically choose an ordering.

Owner-Resolver: Coda [21] relaxes the single-owner scheme described above, in allowing data to be replicated throughout a network, and for changes to be made to the replicas. Coda's focus is on allowing updates in the presence of network partition: nodes might need to make changes without having access to the primary copy. Once connectivity is restored, the newly modified replica must be reconciled with the original data and any other changed replicas; Coda does this by sharing and replaying logs of changes made to the different replicas. If Coda determines that multiple concurrent changes were made, then activates an application-specific conflict resolver that attempts to resolve the conflicts. In the worst case, the data may need to be branched.

There are two design choices that greatly influence the implementation of a shared virtual memory: the granularity of the memory units (i.e., the "page size") and the strategy for maintaining coherence.

Distributed File System

A distributed file system is a resource management component of a distributed operating system. It implements a common file system that can be shared by all the autonomous computers in the system.

The file system is designed to help programmers manage their local naming environments and share consistent versions of collections of software[7]. It names multiple versions of local and remote files in a hierarchy. Local names can refer to local files or be attached to remote files. Remote files also may be referred to directly. Remote files are immutable and cached on the local disk.

Two important goals of distributed file systems follow:

Network teansparency: the primary goal of a distributed file system is to provide the same functional capabilities to access files distributed over a network as the file system of a timesharing mainframe system does to access files residing at one location. Ideally users do not have to be aware of the location of files to access them. This property of a distributed file system is known as network transparency[16].

High availability: Another major goal of distributed file system is to provide high availability. Users should have the same easy access to files, irrespective of their physical location. System failures or regularly scheduled activities such as backups or maintenance should not result in the availability of files.

Architecture of Distributed File System

Ideally in a distributed file system, files can be stored at any machine and the computation can be performed at any machine. When a machine needs to access a file stored on a remote machine, the remote machine performs the necessary file access operation and returns data if a read operation is performed. However, for higher performance, several machines, referred to as *file servers* [4][7]are dedicated to storing files and performing storage and retrieval operations. The rest of the machines in the system can be used solely for computation purposes. These machines are referred to as *clients* [puff][7] and they access files stored on servers. A configuration of personal workstations, each with a local disk, connected to shared file servers by a local area network can provide a responsive base for software development by a team of programmers. The workstations provide each programmer with dedicated hardware resources that respond quickly to interactive demands. The file servers provide a way for the group of programmers to share information A file system that supports a group of cooperating programmers has two important jobs to do. First, it must help each programmer manage a private file naming environment in which to work. Second, it must help the group share consistent versions of the software subsystems being developed in parallel. CFS[7] addresses these requirements by providing each workstation with a hierarchical name space that includes the files on the local disk and on all file servers. . The local files are private to the workstation.

The remote files are sharable among all workstations. The replication control protocol [17] that guarantees consistency in the face of node and network failure. To provide strict or sequential consistency at little cost to exclusive or shared reads. To realize this goal, we use a primary copy method with server redirection when concurrent writes occur. The strategy differs from the usual primary copy scheme in that it allows late and dynamic binding of the primary server, chosen at the granularity of a single file or directory.

IVY: Integrated Shared Virtual Memory At Yale

Ivy is a multi-user read/write peer-to-peer file system[1]. Ivy has no centralized or dedicated components, and it provides useful integrity properties without requiring users to fully trust either the underlying peer-to-peer storage system or the other users of the file system. . Since the first prototype IVY took its breath into life in 1986[13], the development of software DSM systems can be divided into three important phases: **ancient history (1986-1990)**, **renaissance period (1991-1996)**, and **present day (1997-2000)** While designing a peer-to-peer file system [12] following points should be considered:

• Peers have direct control of their resources. Each peer may administer its own storage and file objects and perform operations on them independently of their location and usage in the network.

• Peers have control of how their resources are used. Each peer may authorize specific peers to certain actions. Also each peer may define its own sharing policy.

• Peers should be able to allocate and use resources they do not physically possess. This can be achieved either by pooling of resources or sharing, as long as the process complies with the previous requirements.

• All actions should be accountable. Every transaction in the network should be traceable to a named peer, resource or combination of two.

• The network's capacity should grow as more nodes join it, in typical peer-to-peer fashion. Moreover, well connected and well resourced nodes should be exploited when needed and if they allow so. Moreover, the Grid environment[12] we target has imposed special requirements, including:

• Shared namespaces: In addition to sharing file contents, participants should be able to agree on common collections or clusters [20] of files. This is traditionally achieved through distributed filesystem designs where numerous peers agree on a common namespace of data. We should allow equal functionality, additionally supporting the adhoc creation and management of multiple such views.

• Support for multiple storage types: As we presume cooperation among new and already deployed file services, we should provide mechanisms for merging existing data exported via GridFTP, FTP, HTTP, etc. into the same distributed namespace and allow seamless access to objects disregarding the transfer protocol or location.

• Support for special file types: Data contained in files may have special semantics, and as so require or support special operations beyond access, move, copy, delete, etc. For example log files may provide special mechanisms to append entries or files storing experimental results from scientific measurements may contain special metadata.

An Ivy file system consists solely of a set of logs, one log per participant. Ivy stores its logs in the DHash distributed hash table. Each participant finds data by consulting all logs, but performs modifications by appending only to its own log. This arrangement allows Ivy to maintain meta-data consistency without locking. Ivy users can choose which other logs to trust, an appropriate arrangement in a semi-open peer-to-peer system.

Ivy presents applications with a conventional file system interface. When the underlying network is fully connected, Ivy provides NFS-like semantics[18], such as close-to-open consistency. Ivy detects conflicting modifications made during a partition, and provides relevant version information to application-specific conflict resolvers. Performance measurements on a wide-area network show that Ivy is two to three times slower than NFS.

Ivy presents a single file system image that appears much like an NFS file system[18]. In contrast to NFS, Ivy does not require a dedicated server; instead, it stores all data and meta-data in the DHash peer-to-peer block storage system. DHash can distribute and replicate blocks, giving Ivy the potential to be highly available. One possible application of Ivy is to support distributed projects with loosely affiliated participants. , All peer-to-peer filesystems, except Ivy, manage a single, distributed namespace[12]. Ivy creates a namespace per user and addresses issues like shared namespaces (views), although the corresponding mechanisms are cumbersome as they depend on the read-only nature of the underlying DHT[8]

Building a shared read-write peer-to-peer file system poses a number of challenges.

- First, multiple distributed writers[16] make maintenance of consistent file system meta-data difficult.
- Second, unreliable participants make locking an unattractive approach for achieving meta-data consistency.
- Third, the participants may not fully trust each other, or may not trust that the other participants' machines have not been compromised by outsiders;

Thus there should be a way to ignore or un-do some or all modifications by a participant revealed to be untrustworthy. Finally, distributing file-system data over many hosts means that the system may have to cope with operation while partitioned, and may have to help applications repair conflicting updates made during a partition.

Ivy uses logs to solve the problems described above. Each participant with write access to a file system maintains a log of changes they have made to the file system. Participants scan all the logs (most recent record first) to look up file data and meta-data. Each participant maintains a private snapshot to avoid scanning all but the most recent log entries[9]. The use of per-participant logs, instead of shared mutable data structures, allows Ivy to avoid using locks to protect meta-data. Ivy stores its logs in DHash, so a participant's logs are available even when the participant is not.

Ivy resists attacks from non-participants, and from corrupt DHash servers, by cryptographically verifying the data it retrieves from DHash. An Ivy user can cope with attacks from other Ivy users by choosing which other logs to read when looking for data, and thus which other users to trust. Ignoring a log that was once trusted might discard useful information or critical meta-data; Ivy provides tools to selectively ignore logs and to fix broken meta-data.

Ivy provides NFS-like file system semantics[18] when the underlying network is fully connected. For example, Ivy provides close-to-open consistency. In the case of network partition, DHash replication may allow participants to modify files in multiple partitions. Ivy's logs contain version vectors that allow it to detect conflicting updates after partitions merge, and to provide version information to application-specific conflict resolvers.

The Ivy implementation uses a local NFS loop-back server [18] to provide an ordinary file system interface. Performance is within a factor of two to three of NFS. The main performance bottlenecks are network latency and the cost of generating digital signatures on data stored in DHash.

Following sections describes a read/write peer-to-peer storage system; previous peer-to-peer systems have supported read-only data or data writeable by a single publisher. It describes how to design a distributed file system with useful integrity properties based on a collection of untrusted components. Finally, it explores the use of distributed hash tables as a building-block for more sophisticated systems.

Design

An Ivy file system consists of a set of logs, one log per participant[9]. A log contains all of one participant's changes to file system data and meta-data. Each participant appends only to its own log, but reads from all logs. Participants store log records in the DHash distributed hash system, which provides per-record replication[17] and authentication. Each participant maintains a mutable DHash record (called a *log-head*) that points to the participant's most recent log record. Ivy uses version vectors [9] to impose a total order on log records when reading from multiple logs. To avoid the expense of repeatedly reading the whole log, each participant maintains a private snapshot summarizing the file system state as of a recent point in time.

The Ivy implementation acts as a local loop-back NFS v3 [18] server, in cooperation with a host's in-kernel NFS client support. Consequently, Ivy presents file system semantics much like those of an NFS v3 file server.

DHash

Ivy stores all its data in DHash [8]. DHash is a distributed peer-to-peer hash table mapping keys to arbitrary values. DHash stores each key/value pair on a set of Internet hosts determined by hashing the key. This paper refers to a DHash key/value pair as a DHash block. DHash replicates blocks to avoid losing them if nodes crash.

DHash ensures the integrity of each block with one of two methods. A *content-hash block* requires the block's key to be the SHA-1 cryptographic hash of the block's value; this allows anyone fetching the block to verify the value by ensuring that its SHA-1 hash matches the key. A *public-key block* requires the block's key to be a public key, and the value to be signed using the corresponding private key. DHash refuses to store a value that does not match the key. Ivy checks the authenticity of all data it retrieves from DHash. These checks prevent a malicious or buggy DHash node from forging data, limiting it to denying the existence of a block or producing a stale copy of a public-key block.

Ivy participants communicate only via DHash storage; they don't communicate directly with each other except when setting up a new file system. Ivy uses DHash content-hash blocks to store log records. Ivy stores the DHash key of a participant's most recent log record in a DHash block called the log-head[8]; the log-head is a public-key block, so that the participant can update its value without changing its key. Each Ivy participant caches content-hash blocks locally without fear of using stale data, since content-hash blocks are immutable. An Ivy participant does not cache other participants' log-head blocks, since they may change.

Ivy uses DHash through a simple interface: put(key, value) and get(key). Ivy assumes that, within any given network partition, DHash provides write-read consistency; that is, if put(k, v) completes, a subsequent get(k) will yield v. The current DHash implementation does not guarantee write-read consistency; however, techniques are known which can provide such a guarantee with high probability These techniques require that DHash replicate data and update it carefully, and might significantly decrease performance. Ivy operates best in a fully connected network, though it has support for conflict detection after operating in a partitioned network .

Ivy would in principle work with other distributed hash tables, such as CFS [7], Ivy [1], Pond [8], PAST [8], Total Recall [8], and Glacier [8]. All of these systems use consistent hashing (or a variant) to balance load. Some of these systems are designed to improve the availability of *individual o*bjects

Log Data Structure

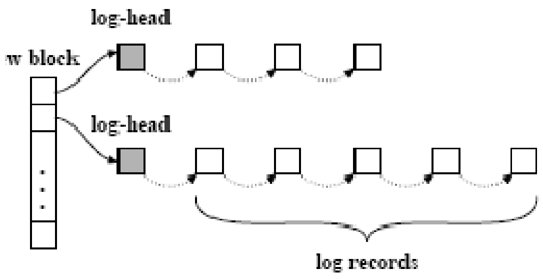

Figure: Example Ivy view and logs. White boxes are DHash content-hash blocks; gray boxes are public-key blocks.

An Ivy log consists of a linked list[9] of immutable log records. Each log record is a DHash content-hash block. Table 1 describes fields common to all log records. The prev field contains the previous record's DHash key. A participant stores the DHash key of its most recent log record in its log-head block. The log-head is a public-key block with a fixed DHash key, which makes it easy for other participants to find.

Table: Fields present in all Ivy log records.

Field	**Use**
prev	DHash key of next oldest log record
head	DHash key of log-head
seq	per-log sequence number
timestamp	time at which record was created
version	version vector

Conflict Resolution

Ivy provides a tool, lc, that detects conflicting application updates to files; these may arise from concurrent writes to the same file by applications that are in different partitions or which do not perform appropriate locking. lc scans an Ivy file system's log for records with concurrent version vectors that affect the same file or directory entry. lc determines the point in the logs at which the partition must have occurred, and determines which participants were in which partition. lc then uses Ivy views to construct multiple historic views of the file system: one as of the time of partition, and one for each partition just before the partition healed. For example,

```
% ./lc -v /ivy/BXz4+udjsQm4tX63UR9w71SNP0c
before: +WzW8s7fTEt6pehaB7isSfhkc68
partition1: l3qLDU5icVMRrbLvhxuJ1WkNvWs
partition2: JyCKgcsAjZ4uttbbtIX9or+qEXE
% cat /ivy/+WzW8s7fTEt6pehaB7isSfhkc68/file1
original content of file1
% cat
/ivy/l3qLDU5icVMRrbLvhxuJ1WkNvWs/file1
original content of file1, changed
append on first partition
% cat /ivy/JyCKgcsAjZ4uttbbtIX9or+qEXE/file1
original content of file1
append on second partition
```

In simple cases, a user could simply examine the versions of the file and merge them by hand in a text editor. Application-specific resolvers such as those used by Coda [13] could be used for more complex cases.

Security and Integrity

Since Ivy is intended to support distributed users with arms-length trust relationships, it must be able to recover from malicious participants. The situation we envision is that a participant's bad behavior is discovered after the fact. Malicious behavior is assumed to consist of the participant using ordinary file system operations to modify or delete data. One form of malice might be that an outsider breaks into a legitimate user's computer and modifies files stored in Ivy.

To cope with a good user turning bad, the other participants can either form a new view that excludes the bad participant's log, or form a view that only includes the log records before a certain point in time. In either case the resulting file system may be missing important meta-data. Upon user request, Ivy's ivycheck tool will detect and fix certain meta-data inconsistencies. ivycheck inspects an existing file system, finds missing Link and Inode meta-data, and creates plausible replacements in a new *fix log*. ivycheck can optionally look in the excluded log in order to find hints about what the missing meta-data should look like.

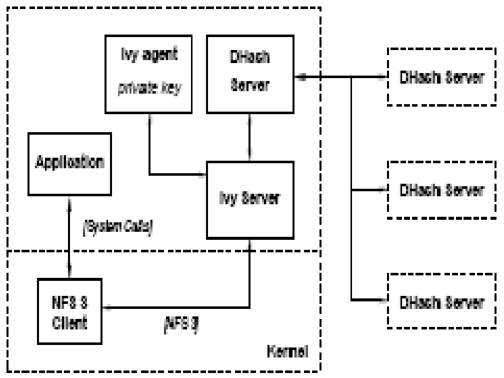

Figure 4: Ivy software structure.

Related Work

Ivy was motivated by recent work on peer-to-peer storage, particularly FreeNet , PAST[19] , and CFS [7,13]. The data authentication mechanisms in these systems limit them to read-only or single-publisher data, in the sense that only the original publisher of each piece of data can modify it. CFS [13]builds a file-system on top of peer-to-peer storage, using ideas from SFSRO ; however, each file system is read-only. Ivy's primary contribution relative to these systems is that it uses peer-to-peer storage to build a read/write file system that multiple users can share. The first heterogeneous distributed shared memory prototype named Mermaid [13] was designed and implemented by Songnian Zhou et.al. Mermaid was implemented on the IVY DSM system and supports C language. The initial DSM algorithm [20] is a simple sequentially consistent, multiple-reader/single-writer algorithm, based on that used in IVY and other systems. The machine pages of the virtual machine are divided between the nodes, such that each node manages a subset of the pages. When a node faults on a page, the manager
node is contacted in the first instance. The manager node then forwards to the owner (if it is not itself the owner), and the owner returns the data directly to the requesting node. The copyset is sent along with the data, and if necessary the receiving node performs any invalidations. Version numbers are used to avoid re-sending unchanged page data.

Log-structured File System

Sprite LFS [15] represents a file system as a log of operations, along with a snapshot of i-number to i-node location mappings. LFS uses a single log managed by a single server in order to to speed up small write performance. Ivy uses multiple logs to let multiple participants update the file system without a central file server or lock server; Ivy does not gain any performance by use of logs.

Distributed Storage Systems

Zebra maintains a per-client log of file contents, striped across multiple network nodes. Zebra serializes meta-data operations through a single meta-data server. Ivy borrows the idea of per-client logs, but extends them to meta-data as well as file contents. This allows Ivy to avoid Zebra's single meta-data server, and thus potentially achieve higher availability.

xFS [3], the Serverless Network File System, distributes both data and meta-data across participating hosts. For every piece of meta-data (e.g. an i-node) there is a host that is responsible for serializing updates to that meta-data to maintain consistency. Ivy avoids any meta-data centralization, and is therefore more suitable for wide-area use in which participants cannot be trusted to run reliable servers. However, Ivy has lower performance than xFS and adheres less strictly to serial semantics.

Frangipani [19] is a distributed file system with two layers: a distributed storage service that acts as a virtual disk and a set of symmetric file servers. Frangipani maintains fairly conventional on-disk file system structures, with small, per-server meta-data logs to improve performance and recoverability. Frangipani servers use locks to serialize updates to meta-data. This approach requires reliable and trustworthy servers.

Harp [19] uses a primary copy scheme to maintain identical replicas of the entire file system. Clients send all NFS requests to the current primary server, which serializes them. A Harp system consists of a small cluster of well managed servers [20], probably physically co-located. Ivy does without any central cluster of dedicated servers--at the expense of strict serial consistency.

Pastis [22], a completely decentralized multi-user read-write peer-to-peer _le system. Pastis' design is simple compared to other existing systems, as it does not require complex algorithms like Byzantine-fault tolerant (BFT) replication or a central administrative authority. It is also highly scalable in terms of the number of network nodes and users sharing a given _le or portion of the _le system. Furthermore, Pastis takes advantage of the fault tolerance and good locality properties of its underlying storage layer, the Past DHT.

Keso[13], a distributed and completely decentralized file system based on the peer-to-peer overlay network DKS. In the system we looked at there was three times as much storage space available on workstations than was stored in the distributed file system. The main goals for the design of Keso has been that it should make use of spare resources, avoid storing unnecessarily redundant data, scale well, be self-organizing and be a secure file system suitable for a real world environment. By basing Keso on peer-to-peer techniques it becomes highly scalable, fault tolerant and self-organizing. Keso is intended to run on ordinary workstations and can make use of the previously unused storage space. Keso also provides means for access control and data privacy despite being built on top of untrusted components. The file system utilizes the fact that a lot of data stored in traditional file systems is redundant by letting all files that contains a datablock with the same contents reference the same datablock in the file system. This is achieved while still maintaining access control and data privacy.

Reclaiming Storage

The Elephant file system allows all file system operations to be undone for a period defined by the user, after which the change becomes permanent. While Ivy does not currently reclaim log storage, perhaps it could adopt Elephant's version retention policies; the main obstacle is that discarding log entries would hurt Ivy's ability to recover from malicious participants. Experience with Venti suggests that retaining old versions of files indefinitely may not be too expensive.

Consistency and Conflict Resolution

Coda [13] allows a disconnected client to modify its own local copy of a file system, which is merged into the main replica when the client re-connects. A Coda client keeps a replay log that records modifications to the client's local copies while the client is in disconnected mode. When the client reconnects with the server, Coda propagates client's changes to the server by replaying the log on the server. Coda detects changes that conflict with changes made by other users, and presents the details of the changes to application-specific conflict resolvers. Ivy's behavior after a partition heals is similar to Coda's conflict resolution: Ivy automatically merges non-conflicting updates in the logs and lets application-specific tools handle conflicts.

Ficus [9] is a distributed file system in which any replica can be updated. Ficus automatically merges non-conflicting updates from different replicas, and uses version vectors to detect conflicting updates and to signal them to the user. Ivy also faces the problem of conflicting updates performed in different network partitions, and uses similar techniques to handle them. However, Ivy's main focus is connected operation; in this mode it provides close-to-open consistency, which Ficus does not, and (in cooperation with DHash) does a better job of automatically distributing storage over a wide-area system.

Bayou [9] represents changes to a database as a log of updates. Each update includes an application-specific *merge procedure* to resolve conflicts. Each node maintains a local log of all the updates it knows about, both its own and those by other nodes. Nodes operate primarily in a disconnected mode, and merge logs pairwise when they talk to each other. The log and the merge procedures allow a Bayou node to re-build its database after adding updates made in the past by other nodes. As updates reach a special primary node, the primary node decides the final and permanent order of log entries. Ivy differs from Bayou in a number of ways. Ivy's per-client logs allow nodes to trust each other less than they have to in Bayou. Ivy uses a distributed algorithm to order the logs, which avoids Bayou's potentially unreliable primary node. Ivy implements a single coherent data structure (the file system), rather than a database of independent entries; Ivy must ensure that updates leave the file system consistent, while Bayou shifts much of this burden to application-supplied merge procedures. Ivy's design focuses on providing serial semantics to connected clients, while Bayou focuses on managing conflicts caused by updates from disconnected clients.

Storing Data on Untrusted Servers

BFS [15], OceanStore [15,16], and Farsite [13] all store data on untrusted servers using Castro and Liskov's practical Byzantine agreement algorithm [15]. Multiple clients are allowed to modify a given data item; they do this by sending update operations to a small group of servers holding replicas of the data. These servers agree on which operations to apply, and in what order, using Byzantine agreement. The reason Byzantine agreement is needed is that clients cannot directly validate the data they fetch from the servers, since the data may be the result of incremental operations that no one client is aware of. In contrast, Ivy exposes the whole operation history to every client. Each Ivy client signs the head of a Merkle hash-tree [9]of its log. This allows other clients to verify that the log is correct when they retrieve it from DHash; thus Ivy clients do not need to trust the DHash servers to maintain the correctness or order of the logs. Ivy is vulnerable to DHash returning stale copies of signed log-heads; Ivy could detect stale data using techniques introduced by SUNDR Ivy's use of logs makes it slow, although this inefficiency is partially offset by its snapshot mechanism.

TDB, S4, and PFS use logging and (for TDB and PFS) collision-resistant hashes to allow modifications by malicious users or corrupted storage devices to be detected and (with S4) undone; Ivy uses similar techniques in a distributed file system context.

Spreitzer et al. suggest ways to use cryptographically signed log entries to prevent servers from tampering with client updates or producing inconsistent log orderings; this is in the context of Bayou-like systems. Ivy's logs are simpler than Bayou's, since only one client writes any given log. This allows Ivy to protect log integrity, despite untrusted DHash servers, by relatively simple per-client use of cryptographic hashes and public key signatures.

Conclusion

This seminar report presents Ivy, a multi-user read/write peer-to-peer file system. Ivy is suitable for small groups of cooperating participants who do not have (or do not want) a single central server. Ivy can operate in a relatively open peer-to-peer environment because it does not require participants to trust each other.

An Ivy file system consists solely of a set of logs, one log per participant. This arrangement avoids the need for locking to maintain integrity of Ivy meta-data. Participants periodically take snapshots of the file system to minimize time spent reading the logs. Use of per-participant logs allows Ivy users to choose which other participants to trust.

Due to its decentralized design, Ivy provides slightly non-traditional file system semantics; concurrent updates can generate conflicting log records. Ivy provides several tools to automate conflict resolution. More work is under way to improve them.

Also the distributed file system architecture is explained along with all the features of distributed file system. The virtual shared memory concept is also explained.

References

[1] Muthitachareon, R Morris, T. M. Gil,B. Chen. *Ivy: A Read/Write Peer-to-Peer File System*. In Proceding of 5th Symposium on Operating System Design and Implementation (OSDI 2002)

[2] Kai li, Paul Hudak. *Memory Coherence in Shared Virtual Memory Systems*.

[3] M. Singhal, N. Shivratri. *Advanced concepts in Operating Systems* page199-200.

[4] Antti Kantee. *Send and Receive of File System Protocols: Userspace Approch with puffs*. In Helsinki University of Technology.

[5] A. Chazapis, A. Zissimos, N. Koziris. *A peer-to-peer replica management service for high throughput Grids*.

[6] M. Singhal, N. Shivratri. *Advanced concepts in Operating Systems* page 248-252.

[7] Michael D. Schroeder, David K. Gifford and Roger M. Needham Xerox Pals Alto Research Center : *A Caching File System For a Programmer's Workstation*

[8] Jeffrey Pang , Phillip B. Gibbons, Michael Kaminsky, Srinivasan Seshan, Haifeng Yu..Carnegie Mellon University ,Intel Research Pittsburgh National University of Singapore*: Defragmenting DHT-based Distributed File Systems*

[9] Martin Placek And Rajkumar Buyya The University of Melbourne: *A Taxonomy of Distributed Storage Systems*

[10] S. Zhou, T. McInerney, M. Snelgrove, M. Stumm, D. Wortman. Computer system research institute, University of Torento: *Shared Virtual Memory: A Simple Model For Implementing Distributed Applications*

[11] Andrew S. Tanenbaum ,Henri E. Bal .Dept. of Mathematics and Computer Science, Vrije Universiteit Amsterdam, The Netherlands, M. Frans Kaashoek, Laboratory for Computer Science, M.I.T. Cambridge, MA *: Programming a Distributed System Using Shared Objects.*

[12] Antony Chazapis, Georgios Tsoukalas, Georgios Verigakis, Kornilios Kourtis, Aristidis Sotiropoulos and Nectarios Koziris ,National Technical University of Athens, School of Electrical and Computer Engineering, Computing Systems Laboratory: *Global-scale peer-to-peer file services with DFS*

[13] Weisong Shi, Department of Computer Science, Courant Institute of Mathematics Sciences, New York University *: Heterogeneous Distributed Shared Memory on Wide Area Network.*

[14] M. Amnefelt, J. Svenningsson *: Keso- A Scalable, Reliable And Secure Read/Write Peer-To-Peer File System [2004]*.

[15] Fabio Picconi, Jean-Michel Busca, Pierre Sens LIP6, Universit´e Paris 6 - CNRS, Paris, France INRIA, Rocquencourt, France : *Exploiting Network Locality in a Decentralized Read-write Peer-to-peer File System.*

[16] Gabriel Antoniu, Luc Bougé, Mathieu Jan , Projet Paris— Septembre 2003 : *Peer-to-Peer Distributed Shared Memory?*

[17] Chaitanya Vinay Hazarey, CSCI 555 Term Research Paper, Department of Computer Science, University of Southern California, Los Angeles : *Unlearning the Traditional File Systems Model, Applying Peer to Peer Techniques for Information Management*

[18] Jiaying Zhang ,Peter Honeyman CITI Technical Report 04-01 *: Replication Control in Distributed File Systems*

[19] Kevin Fu and M. Frans Kaashoek, Massachusetts Institute of Technology and David Mazie`Res, New York University : *Fast and Secure Distributed Read-Only File System[2002]*

[20] Siddhartha Annapureddy, Michael J. Freedman, David Mazi`eres, New York University : *Shark: Scaling File Servers via Cooperative Caching*

[21] Matthew Chapman and Gernot Heiser, *The University of New* South Wales, Sydney, Australia National ICT Australia, Sydney, Australia : *Implementing Transparent Shared Memory on Clusters Using Virtual Machines*

[22] Zachary Ives, Computer and Information Science Department, University of Pennsylvania Philadelphia, *PA 19104-6389: Updates And Transactions In Peer-To-Peer Systems*