Issues in Logging Techniques through a Study of Four Systems

MICHEL RUFFIN
University of Glasgow, Department of Computing Science, Glasgow G12 8RZ, Scotland
INRIA – Projet SOR, Rocquencourt, France

Logging is a particular storage semantic involving specific storage mechanisms, e.g. finding the log end or sequencing through a log. This paper presents the main logging techniques issues through a study of four existing logging systems implemented as independent storage systems: Clio, Camelot DLF, $\mathrm{K_I{}^TL^OG}$ and Sprite LFS. It aims to provide a clear view of problems and actual solutions.

Key words and phrases: Logging techniques, compaction, log end, quick data lookup.

Introduction

An increasing number of applications use logging for multiple purposes [Daniels 1990, Satyanarayanan et al 1992a, Satyanarayanan et al 1992b, Ruffin 1994]. Surprisingly, very few logging facilities exist. Moreover, most of them are dedicated to a particular use. Since little research has been done in this domain, logging techniques are not very well known.

Logging techniques are characterised by the append-only semantics of storage. This semantic is adapted to support fault-tolerant systems and produces an efficient writing speed bringing new challenges.

This paper presents the main issues when designing a logging system. It examines the problems of logging and the solutions brought by some existing logging systems implemented as self-sufficient storage systems. It compares the specifics of these stores: interface, storage organisation, storage compaction, log end lookup, and data lookup. Although, comparison is not always feasible due to the diversity of the systems which have been designed for various purposes.

The logging systems studied are: Clio [Finlayson and Cheriton 1987, Finlayson 1989] a logging service dedicated to file system implementation but general enough to be used for other logging purposes; Camelot DLF [Daniels et al. 1987, Daniels 1988], a logging service designed for transactions; $\mathrm{K_I{}^TL^OG}$ [Ruffin 1992a, Ruffin 1992b], a general-purpose logging system and Sprite LFS [Douglis and Ousterhout 1989, Rosenblum and Ousterhout 1991], a Log-Structured File System. These logging systems were chosen because: they are self-sufficient storage systems, they present important differences and they are relatively well documented.

Section 1 presents the log concept. Section 2 gives an overview of the systems under study, presenting their specifics, their storage units and their interface. Section 3 presents the physical storage organisations on which the basic logging mechanisms rely: log compaction (Section 4), finding the log end (Section 5) and quick data lookup (Section 6). Figure 1 gives the sections where the different systems are described.

Feature	Clio	Camelot	$\mathrm{K_I{}^TL^OG}$	Sprite
Overview	2.2	2.1	2.3	2.4
Organisation	3.1	3.2	3.3	3.4
Compaction	–	4.3.1	4.3.2	4.3.3
Log end	–	5.2	5.3	5.1
Data lookup	6.1	6.2	6.3	–

Figure 1: Sections of systems description

This work was partly supported by grant number ERBCHBGCT92009 of the HCM European Union program at the University of Glasgow and partly by Esprit Basic Research Action Broadcast 6360 at INRIA.

1 Logs

A log is an append-only write, sequential read store. In many implementations, a log is a plain file where data are appended sequentially to the end of the file. When a problem arises in the system (e.g. a fault or an intrusion), the log is reread to find its source and/or to correct its consequences.

1.1 Record

The log storage unit is the *record*. A **record** is composed of a *body* and a *descriptor*. The **body** is the information that the logging client wishes to store. It is a data block of any size, opaque for the logging system. The **descriptor** may contain information to add semantics to the record or to find its location on the log or its origins.

1.2 Log definition

A **log** has the following storage semantics. Writes append records to the end of the log in arrival order. Some records may become obsolete; they will eventually be removed to re-use free space. Other writes are forbidden. Reads may be random, but are mostly sequential. A **backward read** starts at the end of the log and is sequential in reverse storage order. A **forward read** starts at some point of the log and continues in storage order.

A **flush** operation forces data writes onto the storage medium (since generally records are buffered for efficiency). When this operation returns, all the data buffered in memory are guaranteed written on their final storage medium.

1.3 Logical and physical logs

We distinguish two kind of logs: *logical* and *physical* logs.

A **logical log** is the virtual resource of a log seen by a client program. It is composed of a set of records logically linked together. The link semantic is only relevant for the client.

A **physical log** is the concrete storage containing records from several logical logs that are multiplexed onto the same storage medium.

The concepts of logical and physical log coincide when a physical log contains all the records of a single logical log.

2 Overview of the systems studied

This section presents the main characteristics of the four systems studied as well as their record and logical log units and their interfaces.

2.1 Overview of Camelot DLF

The Camelot Distributed Logging Facility (DLF) [Daniels et al. 1987, Daniels 1988] is a logging service designed for transaction processing [Spector 1987]. The service clients are recovery managers.

One Camelot's distinguishing characteristics is its replication algorithm. Log records are replicated for reliability and availability. Suppose there are N log servers on different sites. The log records are replicated on W log servers. The servers replicating a particular client log may change with time; consequently records of a client log are not necessary all stored by the same set of W servers. A log containing all records is available with a set of $N - W + 1$ servers[1] alive and connected by a network (see [Daniels 1988] for details). Therefore, the logging system can tolerate up to $W - 1$ simultaneous site failures.

2.1.1 Record and logical log

A record in Camelot DLF is identified by a `LSN` (Log Sequence Number), an epoch

[1] Since each record is replicated at least on W servers, $N - W$ servers at most do not store the record; consequently a request to $N - W + 1$ servers finds at least one server replicating the record.

`Open()`	Opens the client logical log.
`PutRestartRecord()`	Writes an emergency restart record.
`GetRestartRecord()`	Reads the last emergency restart record.
`Write()`	Appends a new record to the log.
`Force()`	Appends a new record to the log, and writes the record and any previously buffered records on physical storage.
`ForceLSN()`	Writes to physical storage all records with `LSN` less than or equal to a particular `LSN`.
`CreateCursor()`	Creates a cursor on the log.
`CloseCursor()`	Closes a cursor.
`Read()`	Reads a record by `LSN`.
`Next()`	Returns the next (or previous) record.
`HighLsn()`	Returns the highest `LSN` in the logical log.
`HighStableLsn()`	Returns the highest `LSN` on physical storage.
`Truncate()`	Sets the lowest `LSN` to a new value.

Figure 2: Camelot DLF interface

number and a client identifier. `LSN`s and epoch numbers are relative to a particular client. Two records issued successively from the same client will have increasing `LSN`s differing by one. The epoch number is incremented after each client failure.

All the record semantics are not known by the logging service. But two kind of records are distinguished by the system: standard records and "emergency restart records", which are needed for recovering after a failure.

All the clients share the same distributed physical log, but since records are identified relatively to clients, the set of one client's logs is similar to a logical log.

2.1.2 Camelot DLF Interface

Figure 2 presents the Camelot DLF interface (for simplification parameters are omitted).

A `Write()` operation adds a regular record to the log; whereas `PutRestartRecord()` adds an "emergency restart record" to the log. `GetRestartRecord()` permits to read the last "emergency restart record" of the log.

`Write()` stores a standard record in memory; it is flushed latter to the physical medium by `ForceLSN()`. These two operations may be combined in a single step with `Force()`.

`CreateCursor()` creates a "logical log cursor" in order to read records sequentially in a logical log. Once the cursor is created, an initial record can be read with the random read operation `Read()` by providing the `LSN` of the record. Then, the next or previous records can be read with `Next()`. The read direction is a parameter of `CreateCursor()`.

`HighLsn()` and `HighStableLsn()` get the `LSN` of the last record in memory or on the physical medium. The `LSN` of the last record in memory is that of the last record of the logical log, but the `LSN` of the last record on the physical medium may be older, because of buffering.

Finally `Truncate()` tells the logging service that all records with an `LSN` older than a particular `LSN` can be discarded.

2.2 Overview of Clio

Clio [Finlayson and Cheriton 1987, Finlayson 1989] introduced the concept of logical logs under the term *log files*.[2] This system takes advantage of write-once storage media.

[2] For a purpose of clarity, when possible the different abstractions of the systems studied are described with an homogeneous vocabulary instead of using the systems terminologies.

`CreateUio()`	Creates or open a logical log.
`ReleaseUio()`	Completes the updates performed on a logical log e.g. closes or flushes a logical log. (the action of this operation is controled by flags).
`QueryUio()`	Provides information on a logical log.
`ReadUio()`	Reads a record.
`WriteUio()`	Appends a record to a logical log.

Figure 3: Clio's interface

Clio provides a file system service in which files are implemented as logical logs.

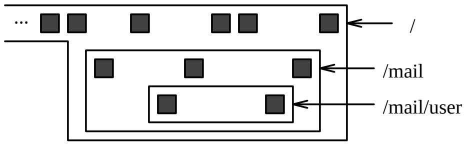

Figure 5: Sub-logs

2.2.1 Record and logical log

Logical logs may be logically copied or can be structured hierarchically. When a logical log is logically copied (see Figure 4), records may be added either to the original or to the logical copy. The two resulting logical logs only share the records that belonged to the original logical log before the copy. Logical copy is possible because Clio uses write-once storage; records are never modified or removed once written.

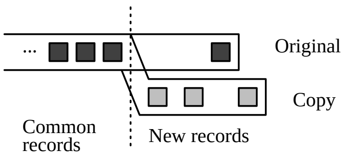

Figure 4: Logical log copy

A logical log may also be created as a sub-log of another one. In this case, the records of the sub-log belong to both logs. Figure 5 shows an example of sub-logs. The logical log "/" contains all the records. The logical log "/mail" contains records corresponding to the mail received by a system. The logical log "/mail/user" provides access to the mail of a particular user.

Each record is identified with a logical log identifier and a time-stamp: the time at which the record has been received by the logging service. For each directory of the Clio file system, a particular log file, called a *catalog* logical log, stores the attributes of the directory files (e.g. file name, ownership or access permission).

2.2.2 Clio's Interface

The Clio interface (see Figure 3) supports a subset of the system V uniform I/O interface [Cheriton 1987] which extends the Unix I/O interface to support locking, transactions and replication.

Cheriton's interface is very general and has been restricted in Clio to an append-only and write-once semantics. For instance the `WriteUio()` operation in its original definition permits to write randomly to an UIO object. In Clio, this operation ignores the block parameter indicating where the write has to be done and appends records at the end of the logical log. Only the five operations presented by Figure 3 have been implemented (see [Cheriton 1987] for more details).

2.3 Overview of K$_{\rm I}$$^{\rm T}L^{\rm O}$$_{\rm G}$

K$_{\rm I}$$^{\rm T}L^{\rm O}$$_{\rm G}$ [Ruffin 1992a, Ruffin 1992b] is a general-purpose logging service which main originality is its ability to be customised to the client needs. The storage functions are decomposed into five orthogonal mechanisms: buffering, remote access, replicating, sharing a logical log, and managing a physical medium. Each such mechanism is embodied into a class of object. For each mechanism,

`Create_log ()`	Creates a logical log.
`Destroy_log ()`	Destroys a logical log.
`Get_lid ()`	Gets a logical log identifier.
`Open_log ()`	Opens a logical log.
`Close_log ()`	Closes a logical log.
`Put ()`	Appends a record to a logical log.
`Get_last ()`	Gets the last record of a logical log.
`Get_prev ()`	Gets the previous record of a logical log.
`Get ()`	Gets the first record of a logical log or a particular record of a logical log.
`Get_next ()`	Gets the next record of a logical log.
`Flush ()`	Forces the writes on physical media.
`Invalidate_records ()`	Invalidates some records.

Figure 6: $\mathrm{K_I{}^TL^OG}$'s interface

multiple policy implementations (or classes) can be provided. Instances of these classes are stackable in any appropriate number or order. A client customises his logical log to a particular set of failure assumptions by selecting adequate classes, instantiating them and connecting the instances together.

2.3.1 Record and logical log

Logical logs have identifiers (`Lid`s) that are unique within the $\mathrm{K_I{}^TL^O{}_G}$ system. Records are identified by `Rid`s that are unique within a logical log and may be either generated by the client or the logging service. `Rid`s generated successively have increased values. Logical logs may be shared or not by several clients.

2.3.2 $\mathrm{K_I{}^TL^O{}_G}$'s Interface

Access to logical logs are done via the interface of a special object called a logging view. The logging view interface is illustrated by Figure 6. It is similar to the Camelot DLF interface.

At creation time, a client may register its logical log with its own identifier (called a client identifier). Given a client identifier, the `get_lid()` operation returns the corresponding `Lid`. A cursor is automatically created when opening a logical log. The four other `get()` operations permits to get records randomly, forward or backward, within a given logical log. The `put()` operation appends a record to a logical log. The `flush()` operation forces the write of all buffered records to the physical medium. `Invalidate_records()` has three forms. The first is similar to Camelot DLF's `Truncate()` operation: all the records of a logical log, before a particular record, are invalidated. The other forms invalidate either a particular record or all the records of a logical log.

2.4 Overview of Sprite LFS

Sprite LFS (Log-structured File System) [Douglis and Ousterhout 1989, Rosenblum and Ousterhout 1991] is a file system structured as a log, in order to take advantage of the speed of the append-only writes.

Files are divided into fixed-size blocks. Those are stored by appending them to the physical log. For each file, an inode structure stores its attributes and points to the file's blocks on the log. The inode is stored in an inode block appended to the log following the file blocks. Inode blocks are pointed to by a table called the inode-map. This table is divided into blocks stored on the device as the table grows. Finally a checkpoint, stored at a fixed position, references the different parts of the inode-map table.

In Sprite LFS, records are blocks. They can be file blocks, blocks storing an inode or blocks storing a part of the inode-map. The logical log concept cannot be clearly identified.

Of the four systems which we are studying, Sprite LFS is the only one that uses fixed-size records and direct pointers to identify its data. Inodes are identified by a unique inode number (an index in the inode-map table), but file blocks are identified by pointers contained in the inode. Therefore, there is no need for record identifiers.

The interface in Sprite LFS is that of a file system.

2.4.1 Discussion

Logical log is a powerful concept. In general-purpose logging systems, logical log can somewhat compensate for the absence of data semantics knowledge. For instance, in a logging system which would be aware of the record semantics of transaction managers, the transaction identifier stored in records could be used to recognise the records of a same transaction. If the transaction is aborted all its records can be discarded from the log; whereas if it commits they have to be archived. By associating a logical log to each transaction, a similar gathering of record can be achieved in a general-purpose logging system. Moreover, hierarchical logical logs provide more facilities to the client of a logging system.

In Sprite LFS, records have no identifier and are referenced by pointers. Whereas in other systems, record have identifiers and no pointer is used. The second solution permits to access records directly (without following pointers) and to move them on the physical log without modifying any pointers. In addition, from the fault-tolerance viewpoint, using pointers can be dangerous: in case of medium failure, a large amount of records may become unreachable.

Apart Sprite LFS, all other log interfaces present a common set of features. A `create()` operation is needed to create or open a logical log. A `write()` operation appends records to the log. A `read()` operation provides backward reads, forward reads and random reads. A `flush()` operation permits to control the storage of record on stable storage. Finally, a `truncate()` operation permits to declare obsolete records.

3 Storage organisation

This section describes and compares the storage organisations, i.e. the physical log structuring of each system. The storage organisations of the systems are determined by their different logging techniques which are described in further sections.

3.1 Clio's organisation

In Clio, the storage medium is divided into fixed-length segments (blocks in Clio's terminology). The segment is the storage addressing unit (in memory or on devices). When a record is written into a segment, a descriptor (called a header in Clio) is appended at the first free location, followed by the record body. The record size is written at the end of the segment (see Figure 7).

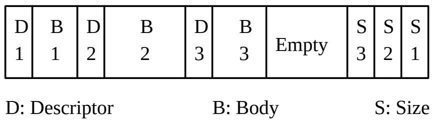

Figure 7: Clio's segment structure

A descriptor is composed of a logical log identifier and the date at which the record was received by the logging service. A record may be split across several segments. Grouping sizes in an index at the end of the segment permits a fast scanning (forward or backward) of the segment contents.

3.2 Camelot's organisation

Camelot also stores records into segments (called blocks in Camelot) of fixed length (see Figure 8). A segment may contain records from different clients. Successive records issued from the same client are gathered by groups. A segment is divided into four areas: a header, the record bodies, the group descriptors and a record index.

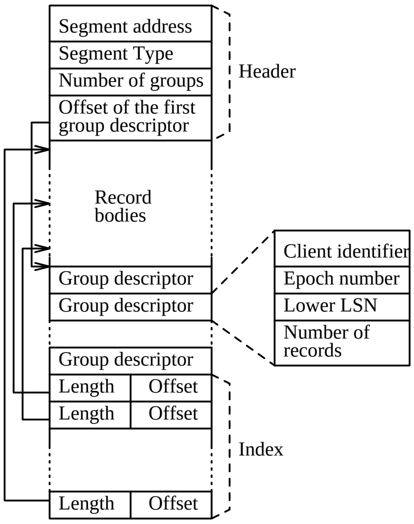

Figure 8: Camelot's segment structure

The header contains the logical segment address, its type, the number of groups in the segment and the offset of the first group.

The logical segment address is for space reuse (see 4.3.1). The physical address of a segment is the logical address modulo the size of the medium. Five types of segment exist depending on their contents: empty segment, data segment or checkpoint header segment, continuation data segment and continuation checkpoint segment.

Groups permit to reduce the meta-data needed to identify records. They gather up to 256 records issued by the same client with a constant epoch number. Since successive `LSN`s issued by a common client are consecutive numbers, a set of successive records created by the same client may be identified with the lowest `LSN`, the number of records, the client identifier and the epoch number.

Finally, the record index at the segment end gives the location of record bodies within the segment.

3.3 K$_{\rm I}$T$^{\rm T}$L$^{\rm O}$G's organisation

Like other logging system, K$_{\rm I}$$^{\rm T}L^{\rm O}$$_{\rm G}$ organises its storage into fixed-length segments (called sectors in K$_{\rm I}$$^{\rm T}L^{\rm O}$$_{\rm G}$). When a record is written (see Figure 9), the record body (B) is written to the first free space in the segment. If it overflows, the body is spread over several segments. The record descriptor (D) (called a trailer in K$_{\rm I}$$^{\rm T}L^{\rm O}$$_{\rm G}$), is then written at the end of the last segment that encloses a part of the record body or in the following segment if the space is too small. The descriptor contains a logical log identifier, a record identifier and the record size.

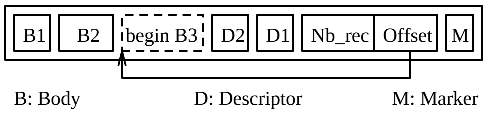

Figure 9: K$_{\rm I}$$^{\rm T}L^{\rm O}$$_{\rm G}$'s segment structure

Each segment is ended by a marker (`M`) used in the algorithms to find the log end and to compact the physical log (detailed latter in §4 and §5). The segment also contains the number of descriptors in the segment (`Nb_rec`) and the offset between the beginning of the segment and the first byte after the end of the last record body whose descriptor is stored in the segment (`Offset`).

These data are sufficient to read the storage medium forward or backward from any segment of the physical log, since segments are stored contiguously on the medium.

Figure 10 presents a storage example in K$_{\rm I}$$^{\rm T}L^{\rm O}$$_{\rm G}$. Starting with the logical log identifier (`Lid`) and the record identifier (`Rid`) of record Rd, its body (Bd) may be found by reading segments 2 and 1. Finding segments 2 and 1 is another problem which will be explained in §6.

In segment 2, `Nb_rec` indicates two valid descriptors. The size of each record body is obtained by examining the descriptors (Dd and De). The location of the end of Rd's body, Bd, is calculated by subtracting the size of body Be from the offset of the segment (minus one byte). The beginning of body Bd is found by subtracting its size (modulo the segment size).

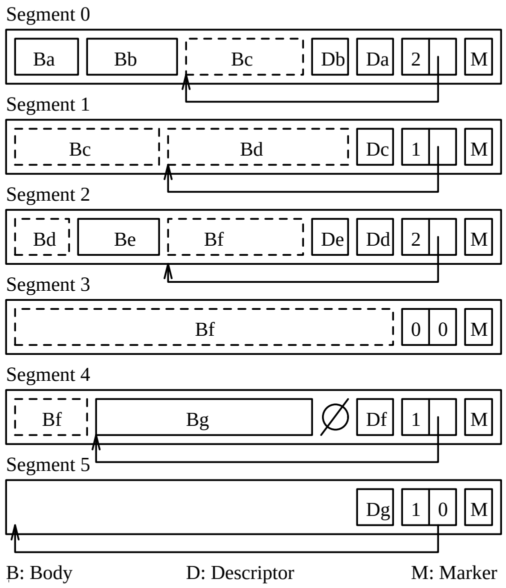

Figure 10: Storage example in $\mathrm{K_{I}^{T}L^{O}_{G}}$

This figure illustrates the main cases that can occur when storing records into segments. Dashed body records are those spread on several segments. In segment 4, the symbol Ø represents an empty space too small for the `Dg` descriptor

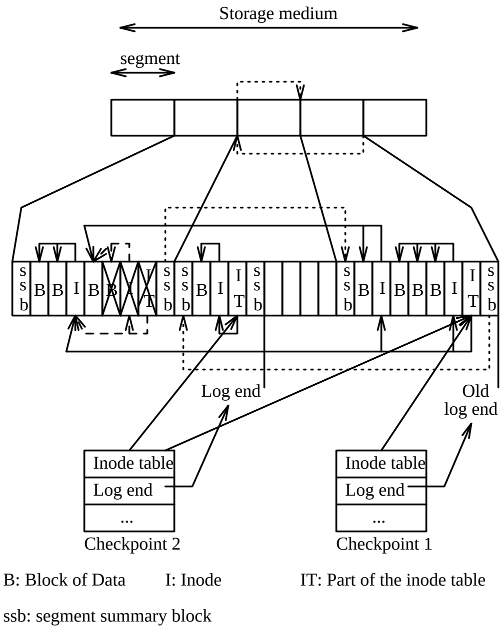

Figure 11: Storage organisation in Sprite LFS

3.4 Sprite LFS organisation

Figure 11 presents the storage organisation in Sprite LFS. The storage medium is organised as large fixed-size segments of a few hundred kilobytes. Segments are threaded in a linked list. Each segment is composed of a set of contiguous, fixed-size blocks (≈ 1 kilobyte). A file is split in data blocks before being written to the physical log. Its inode is appended to the next block. File modification writes the modified data blocks at the end of the log, and appends a new copy of the inode. Inodes are referenced by an inode table stored in blocks which are spread on the physical log. Each block of this table contains the references to the previously written or re-written inodes. The location of the different parts of the inode table is enclosed in a checkpoint block. Writing a new version of the inode table validates the new inodes and hence data. When the checkpoint block is written, the new data becomes permanent. In addition, the checkpoint contains the location of the physical log's end.

At regular intervals, the checkpoint is alternatively written to two fixed positions, in order to tolerate loss of one checkpoint. In case of a failure, data loss is limited to data logged since the last checkpoint. The figure shows two successive states of the log after writing checkpoints.

Segments also contain meta-data "segment summary blocks". `SSB`s link segments together and mark valid data within a segment.

3.5 Commonalities

Four kinds of meta-data can be distinguished: meta-data for record recognition (identification, record size, pointers), meta-data for finding data on physical storage (`Nb_rec` and `Offset` in $\mathrm{K_{I}^{T}L^{O}_{G}}$, `SSB` in Sprite LFS), meta-data for logical log management (owner, access mode), and meta-data for the implementation of physical log management algorithms, such as compaction, finding the log end or fast record lookup (markers, allocation table, ...).

The first kind of meta-data is relatively constant across the different systems: header and size in Clio, trailers in $\mathrm{K_IT L^O G}$, `LSN`s, epoch numbers and information for record location in Camelot. The Sprite LFS case is special, since it is difficult to isolate these meta-data from the other. In each system except Sprite LFS, these informations enclose a record identifier (`LSN`, client identifier and epoch number in Camelot, logical log identifier and record date in Clio, `Lid` and `Rid` in $\mathrm{K_I{}^TL^O{}_G}$).

A commonality of these storage organisations is a fixed-size storage unit: the segment. This storage unit contains meta-data of the second kind needed to identify its contents (e.g. lengths and offset in Camelot, markers in $\mathrm{K_I{}^TL^O{}_G}$ or `SSB` in Sprite LFS), and is used as a unit for writes or reads on the physical media.

The main goal of segmentation is to split the medium into independent parts allowing to treat (e.g. read, write or compact) each part of the medium independently and atomically. Atomicity and independence permits to easily enforce fault-tolerance in the different logging techniques. Segment independence also permits to "understand" what is stored in the segment (i.e. to find the beginning and end of records) and then to read forward or backward all its valid information without starting from the beginning or end of the physical log.

3.6 Space Overhead

Meta-data impose space overhead. One meta-data item is needed to identify each record and one or two are needed for each storage unit. The size of these meta-data items varies with the technology: for instance, pointer size increases with bus (32 bits or 64 bits) and address space size.

The overhead tied to record identification varies widely between systems. Dedicated systems can take advantage of data semantics to save identification overhead while general-purpose systems must provide general identification. For instance, Camelot DLF uses the `LSN` and epoch number semantics to reduce space overhead, knowing that records are identified relatively to client sessions. The client's epoch number stays constant as long as the client is alive, and is incremented at each client failure. Consequently, for a group of records issued from a common client session, only one version of the epoch number has to be kept for their identification. More general identification approaches, such as those of Clio or $\mathrm{K_I{}^TL^O{}_G}$, are not able to provide this improvement; but as a compensation, these systems allow their clients to choose a part of record identification. Sprite LFS cannot easily be compared to these systems since the meta-data needed for log management are mixed to those for file system management.

3.7 Physical log reliability

The storage organisation defines the level of fault-tolerance of the physical log. Sprite LFS stores critical meta-data (checkpoint) alternatively at two fixed storage positions (see §3.4). Therefore, it cannot resist two simultaneous failures in these areas. Clio, Camelot and $\mathrm{K_I{}^TL^O{}_G}$ do not use any centralised information; they can tolerate the loss of any part of a storage medium without losing it all.

This point is important since loss of data on media (and particularly for disks) generally consists in an isolated loss of one or two contiguous disk blocks after an unsuccessful write; however, the loss of larger chunks of one medium is comparatively rare [Hagmann 1987].

4 Compaction

Whatever size may have the storage medium, it is limited, but a physical log never stops to grow. Therefore, some mechanism is needed to free space. Two techniques exist: compacting and archiving. These techniques are not orthogonal, and are often used as complementary techniques. Archiving consists in copying permanent data to another storage medium which is generally slower and of a larger size than the current physical log medium. Compacting moves data on the medium to reduce fragmentation and to facilitate space re-use. In the remainder, we will focus on compacting which is the more complex of the two mechanisms. Clio does not suffer the problems of compaction by using a write-once storage medium.

4.1 Compaction problems

The first problem of compaction stems from the fact that a strong interaction is needed between the logging service and its clients [Daniel 1988, Ruffin 1992a], because it requires to take into account the data semantics. For instance, in transactional systems records of aborted transactions can be removed, those of committed transactions can be archived and the different versions of a same object can be compacted in a version corresponding to its last consistent version. This is illustrated by Kaunitz and Van Ekert [1984].

The second compaction problem occurs when the medium is full and compaction does not free enough space. A solution consists in asking to the client which records can be removed. This requires a mechanism to do this request to the client.

The cost of compaction is one of the major problems. Logging can be forbidden during compaction or may be done asynchronously with compaction. The medium can be compacted in a single step or by chunks. The compaction cost varies depending on the chosen method.

Finally, the last problem of compaction is fault-tolerance. Data cannot be lost if a failure occurs during compaction. To achieve this goal, the writing phases must be atomic, and between these phases the physical log must contain the whole set of valid records.

4.2 Obsolete record declaration

The way records are declared removable varies depending of the semantic knowledge available to the logging system.

In Sprite LFS, where the logging system recognises different record types (file blocks, inode blocks, inode-map blocks and checkpoints), there is no need for obsolete record declaration. When a file is modified, a new inode is appended to the log. The old inode version and the modified file blocks it was referencing are no more reachable. Unreachable data will be automatically removed in the next compaction phase.

Conversely in $\mathrm{K_I{}^TL^O{}_G}$, where nothing is known about records, the client has to explicitly declares the list of obsolete records. The logging service interface operation `invalidate_records()` (see Figure 6) has three forms that permit to declare a particular record, all the records of a logical log or all the records of a logical log before a particular record. These declarations are taken into account at compaction time.

The Camelot logging facility has an intermediate position. In its interface (see Figure 2), it offers an operation `PutRestartRecord()` which differentiates "emergency restart records" from other records. The logging service knows that, for a particular client, records older than an "emergency restart record" may be discarded. When needed, Camelot DLF has the possibility to ask its clients (recovery managers) for a list of obsolete records. Clients can truncate their logical log with the operation `Truncate()`. This mechanism exists also in QuickSilver [Haskin et al. 1987].

4.3 Compaction methods

Compaction algorithms are very specific to each system since they depend on the storage organisation, the failure assumptions of the service and the level of knowledge on data semantics.

Two main techniques exist for re-using free space in logging systems: copying and threading [Rosenblum and Ousterhout 1991]. In copying, valid data are moved to the head of the log; whereas in threading, free space is directly re-used in place: space is split into fixed size segments which are logically linked together and data are grouped into segments. Since free space is composed of chunks of fixed size segments, there is no fragmentation.

Copying has an important overhead cost since the medium has to be totally read and valid data has to be re-written. On the other hand, threading leads to bad write performances since there are few chunks of large contiguous free space on the medium. In addition, the threading technique use critical meta-data such as the link between segments which need to be highly replicated for fault-tolerance.

4.3.1 Compaction in Camelot DLF

Camelot distributed logging facility uses a threaded technique. Clients truncates logs by giving to the logging system the greater `LSN`, before which records may be removed (by writing an "emergency restart record" or explicitly truncating their log). A segment may be re-used when all records in the segment have been declared obsolete by their owners using one of the above ways of declaration. No more detail is given on compaction in Camelot DLF, but some data are probably needed to quickly allocate free segments.

4.3.2 Compaction in $\mathrm{K_{I}{}^{T}L^{O}{}_{G}}$

In $\mathrm{K_{I}{}^{T}L^{O}{}_{G}}$, compaction is done by moving valid data to the beginning of the medium. Except at the end, the compaction algorithm does not need any synchronisation with logging writes.

During compaction, some segments are read, compacted in memory and new segments are written at a new location on the medium. The physical log is divided into three areas. The first area starts at the beginning of the physical log and contains the new (compacted) segments. The second area which ends at the end of the physical log, contains the segments that are still to be compacted. The third area or *compaction area* is the intermediate one. During compaction, this area is moving from the beginning to the end of the physical log, and its size is growing as space is freed.

At the beginning of compaction, the compaction area is too small to store the new compacted segments without scratching the old versions of these segments. Therefore, the new segments cannot directly be written at their new location, because a fault at writing time could result in the loss of new and old segments. To solve this problem, the new segments are written in a special (free) area at the end of the medium before being appended to the first area. Once the compaction area size is large enough, writes can be proceeded directly.

4.3.3 Compaction in Sprite LFS

Sprite LFS uses a combination of copying and threading. The physical log is threaded into segments (see Figure 11). A set of segments are chosen and copied in memory. Their valid data are compacted out into new clean segments. These segments are written back to the physical log and they are re-linked with the other segments. The compacted segments are marked cleaned, re-written and re-linked.

Some additional information is needed for this compaction algorithm: the list of valid blocks of segments and the file to which blocks belong to, in order to be able to update the file inode. This information is stored in the segment summary block of each segment.

At compaction time, valid blocks are grouped into segments according to their belonging to hot or cold files. Hot files are high accessed files; conversely, cold files are low accessed files. Cold segments contain cold blocks from cold files and do not need to be compacted as frequently as hot segments. This leads to improvements in the compaction algorithm and permits a better choice of the segments to be compacted.

This improvement is only feasible with some knowledge of data semantics. In addition, both re-linking segments of valid data after compaction, and re-arranging data for grouping hot and cold data, involve loss of record sequential storage order. In Sprite LFS, this loss has no importance since record storage order is not used; whereas in other log uses like that of recovery managers, the loss of record storage order cannot be tolerated.[3]

4.4 Comparing compaction with garbage collection

Some problems and solutions for log compaction and garbage collection (GC) are common since log compaction is the work of re-using space of the physical log, whereas GC is the work of automatically re-using space in object heaps. This section compares these two kinds of techniques.

4.4.1 Automatic garbage detection

One of the main problems of log compaction is to determine which records can be removed (see §4.2). As in GC techniques, the *reachability criterion* may be used to define the data which have to be kept at compaction time. In GC techniques, a set of data, the *roots*, are considered as always reachable. The roots hold references to other data which permits to reach them by following the references. The set of reachable data may be determined by following recursively all the references from the roots. At GC time, reachable data are kept and unreachable data are removed.

In **dedicated** logging systems, automatic obsolete record detection is easy. For instance, in Sprite LFS where the file system semantics is mixed to the logging ones, the roots are composed of: the last checkpoint, the valid parts of the inode map created after this checkpoint, and the list of valid inodes created after the last part of the inode map. Starting with the checkpoint, all other valid parts of the inode map can be found. The valid parts of the inode map point to all the valid inodes, which point to all the valid file blocks. Unreachable parts of the log may be discarded.

In **general-purpose** logging systems, two major problems arise for the implementation of automatic obsolete record detection:

- The roots are not known by the logging system. For instance in Camelot, the roots might roughly be composed of the last "emergency restart record" of each client and all the following records of the non-aborted transactions. The emergency restart records are recognised by Camelot DLF which is relatively dedicated; but since the logging system is not able to differentiate records of committed transactions from those of aborted transactions, only a part of the roots can be known.

- The links between data are not known by the logging system. For instance, if we compare two possible $\mathrm{K_{I}{}^{T}L^{O}{}_{G}}$ clients: a recovery manager and a file system like KFS ($\mathrm{K_{I}{}^{T}L^{O}{}_{G}}$ File System [Ruffin 1994]), it clearly appears that a general system cannot cope with the specific data semantics of each client. For a recovery manager, checkpoints (or emergency restart records) have to be distinguished from other records. References between data are represented by transaction identifiers (assuming that each record relative to a transaction contains the transaction identifier) and the semantics of the checkpoint and other record structures must be known to localise transaction identifiers. On the other hand in KFS, inode records have to be distinguished from other records, references are represented by log and record identifiers, and the inode and directory struc-

[3] Some logging systems dedicated to transactional recovery [Hagmann and Garcia-Molina 1991, Keen and Dally 1993] change the record storage order at compaction time, but time-stamps are associated to records to determine their order.

tures need to be known.

GC techniques solve this problem by a language approach, or by guessing which sequence of bytes could be a reference. These solutions are not adequate for a general purpose system which does not suppose any data model; new solutions have to be found. For instance, Wirz and Nett [1993] strongly type records in order to bring more semantics to the logging system; but this solution is still dedicated to recovery.

4.4.2 Parallelisation

In GC techniques, the *mutation* is the modification of the set of references on objects and is equivalent to appending data to the log in logging systems; whereas the *collection* is the *detection* and the *reclamation* of objects which are no more referenced. This corresponds to the compaction phase in logging systems.

Log compaction and GC share a common problem: since the compaction (or collection) is time consuming, there is an important need to execute it in parallel with logging (or mutation). Moreover, since tolerance to site failure is an important need of logging systems,[4] being able to split the compaction into small independent atomic steps which can be intertwined with logging is of high interest.

From the viewpoint of GC, the problem stems from the concurrency of accesses to data which may lead to inconsistencies; whereas from the viewpoint of logging the problem is to split compaction into steps which are failure-atomic and independent from each other.

The concurrency problem is not obvious for GC; on the other hand, the append only semantics of logging solves these two problems. Since data is never modified and writes are localised at the end of the log (except at the end of the compaction where logging has to be stopped a short moment[5] to avoid conflicts), the log compaction can easily be isolated from logging and it is easy to decompose the compaction into incremental steps which compact a part of the log atomically.

4.4.3 Collection algorithm

There are two main basic collection techniques. Those which re-use space in place: reference counting [Collins 1960, McBeth 1963] and mark and sweep [McCarthy 1960] and those which copy data: mark and compact [Cohen and Nicolau 1983] and scavenging [Fenichel and Yochelson 1969]. In addition, incremental techniques [Baker 1978, Dijkstra et al. 1978] are often combined with these techniques to parallelise GC with mutation, whereas generational techniques [Ungar 1984, Liebermann and Hewitt 1983] generally based on scavenging permit to group data for a more clever GC. This classification may be compared to the log compaction one (threading and copying), since threading is a restricted case of re-using space in place. In the reminder of this section, we present the compaction techniques of the systems studied in terms of GC.

Camelot's compaction technique looks like reference counting (RC) or mark and sweep (MS). For each segment, the type field of the segment header indicates if the segment is free or not. This field is not a counter of the number of references like in RC but more a boolean like in MS. The way segments are marked is not very clear, but it seems that it is neither at each time a record is invalidated, like in RC, nor in a single phase that trace the whole log, like in MS. Segment marking follows a client log truncation or the write of an "emergency restart record". Camelot uses threading, consequently, like the two GC techniques, it re-uses space in place.

$K_I{}^T L^O{}_G$'s technique is mostly an incremental technique based on mark and compact. t

[4] Since the GC techniques that we consider are not distributed and are assumed to work in memory, site failures have not to be considered from the GC viewpoint because in such a circumstance the memory content is lost.

[5] The interruption of logging can be reduced to the time needed to execute the last step of compaction.

is incremental, because compaction and logging may be sliced and each step of compaction copies more of the initial amount of data to compact. As in the mark and compact technique, records are moved to the beginning of the medium. With mark and compact technique, the references between objects must be updated when object are moved. This is not the case in $K_I{}^TL^O{}_G$ since records do not hold references between themselves.

Sprite LFS technique is incremental, generational and based on scavenging. It is incremental because at compaction time only a part of the medium is compacted: the compaction algorithm chooses a set of segments which will free a large amount of space. This cannot be compared to GC incremental techniques [Baker 1978, Dijkstra et al. 1978] which split the collection in steps and execute them intertwined with mutation. As in generational techniques, data are grouped by generations. Old data (cold file in Sprite) are reclaimed (compacted) at a lower rate than younger (hotter) ones. Sprite LFS technique is similar to scavenging since compacted data are copied to a new area of the medium (new segments).

Ephemeral logging [Keen and Dally 1993] is a good example of a logging system designed following a pure generational technique for log compaction. Since Ephemeral logging is dedicated to transaction processing, automatic obsolete record detection is possible. Moreover, the generational technique is used to solve the problem of aborting long-lived transactions when the log is full and when old and still valid records prevent to compact the log (see [Keen and Dally 1993] for details).

5 Finding the end of the physical log

Finding the log end is a traditional problem to face when designing a logging organisation. Being unable to find the exact log end means that a certain amount of data has been lost.

5.1 Checkpointing the log end

In Sprite LFS the segment corresponding to the end of the log is referenced by a pointer stored in the checkpoint. The checkpoint is alternatively written to two different fixed positions. The checkpoint is not written each time data is appended at the end of the log, to avoid the problem of moving twice the writing heads each time data is appended to the log; but if a site failure occurs, all the data written on the physical medium since the last write of the checkpoint is lost.

5.2 Zeroing free space

The solution of Camelot and Quicksilver [Haskin et al. 1987] is to initialise the storage device, filling it with zeros. At the end of each compaction phases the newly-freed space is re-initialised to zero. In this solution, the end of the physical log is found by binary search, looking for the last part of the device not filled with zeros. The drawback of this method is the overhead of device initialisation at compaction time. In addition, the possibility of large records filled with zeros increase the complexity of the lookup algorithm.

5.3 Flagging the segments

$K_I{}^TL^O{}_G$ solution is based on the use of markers at the end of segments. A marker is composed of a date, a medium number (an arbitrary number chosen at the first use of the medium), and a compaction counter (incremented at each compaction).

When no compaction has ever been done on this medium, it is divided into two zones (see the top part of Figure 12): a zone starting at the beginning of the medium where data has been stored, and a zone containing garbage (its contents depend of the previous uses of the medium). In the first zone, the markers of consecutive segments have strictly increasing dates; all the markers have the same medium number value, and their compaction counter value is 0.

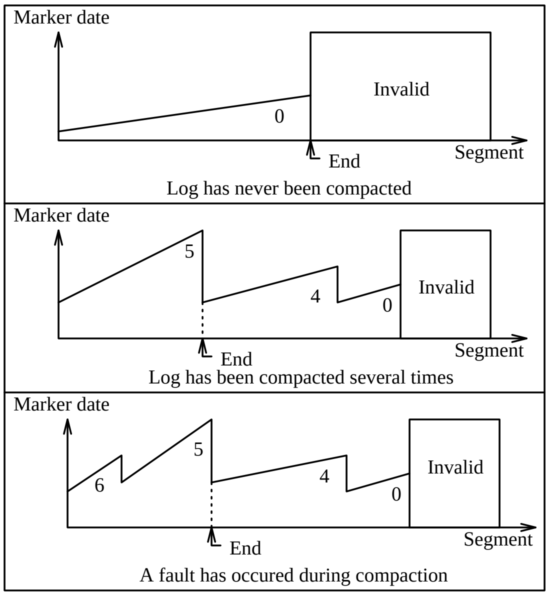

Figure 12: Marker values on the medium

The numbers are the values of the compaction counter. Each compaction zone correspond to a number and a saw-tooth

Finding the log end consists in looking (with a binary search) for the last set of N valid markers on the medium. A valid marker is one that contains the current compaction number and the medium number.[6] A set of markers of consecutive segments are valid if all markers are valid and increasing. The number N determines the probability of error. On a machine with words of 32 bits, with $N = 4$ the probability is less than 10^{-78}.[7]

As we explained in §4.3.2, compaction is done by moving valid records to the beginning of the medium. During compaction, valid records are gathered into new segments. The marker of each new segment is the greatest marker of the compacted segments with an incremented compaction counter. These markers are "not yet valid" until the end of the compaction when the current compaction counter is incremented.

When compaction has been done several times (see the middle of Figure 12), the medium is divided into several zones.[8] Except for the invalid zone (if any), each zone corresponds to a compaction phase (saw-tooth on the Figure). At the end of a compaction phase, zones have the following property: the last marker of a zone is greater or equal to the last marker of the next zone.

The knowledge of the current compaction counter permits to find the limit of the valid zone by a binary search looking for the N last valid marker with the current compaction counter.

When a site fault has occurred during compaction (see the bottom of Figure 12), the zone property is no more valid for the first one. The first zone corresponds to segments which have been moved by compaction. Markers of this zone are not yet valid (their compaction counter is greater than the current one). The second zone is composed of segments that have still to be compacted. The last marker date of the first zone is a copy of one of the second zone.

Consequently at boot time the system look for the end of the two first zones (with binary searches) and by comparing the two last markers, it is able to determine that the medium was under compaction at fault time. To limit this search to the two first zones, the compaction algorithm must be restarted at the point it has been stopped.

5.4 Discussion

The problem of finding the end of the physical log has three components:

[6]The current compaction counter and the medium number may be found by reading the N first segments of the medium. At the first use of the medium these N segments are initialised with markers. At each compaction their compaction counter is incremented.

[7]This probability is calculated with the assumption that the invalid data part is filled with random bits. In other non-malicious cases, when the medium has already been used for logging, the assumption is reduced to non-byzantine failures and the probability is null.

[8]More than two zones can exist, because the physical log may be compacted before it becomes full.

- In most of the cases the log end must be exact. For instance, to support transaction semantics, the `flush()` or `force()` operation guarantees that data have been written stably to their final storage medium. After a failure, the recovered log end position must be after any flushed data.
- Assuming an exact log end, if a checkpoint containing the log end is written to a fixed position of the same storage medium, then at each flush two writes and two seeks are necessary: one write at the end of the log to write data, a seek to the checkpoint position, a write of the checkpoint and a seek to the end. These writes and subsequent head movements lead to poor write performances defeating one of the most important advantage of logging.
- To avoid the problem of moving the writing head at each write, one could write the location of the log end in parallel onto a different device. But writing information that is critical to the understanding of data storage semantics on a fixed position implies, in case of corruption of these data, the loss of the whole set of data on the medium.

To summarise the problem, the system must be able to find the log end exactly after each flush on the medium, but its position should not be written on a fixed position of any storage medium. *The log end should be determinable from "regular" data stored on the log.*

Clio does not suffer the problem of the log end by using write-once media. Sprite LFS is the only of the systems studied which stores the end of the log at regular intervals on a fixed position. Consequently, it is the only one which tolerates a non-exact determination of the log end and does not resist two simultaneous medium failure (in the two checkpoints areas).

QuickSilver, Camelot and $\mathrm{K_I{}^TL^OG}$ determine exactly the end of the log and are able to resist any localised medium failure without losing all the data on the medium. Among these three systems, $\mathrm{K_I{}^TL^O{}_G}$ is the only one which do not format the medium at initialisation and do not need to write all the medium at each compaction.

6 Quick data lookup

In Clio, Camelot and $\mathrm{K_I{}^TL^O{}_G}$ a record can be read by sequentially scanning the physical log forward from the beginning or backward from the end until the record is reached. Special techniques are used to increase record seeks in these systems. These techniques use additional meta-data making a trade off between the space overhead for meta-data and the speed of record seek.

Once again, Sprite LFS cannot be compared to these systems since it uses the file system semantics to achieve this goal.

6.1 Quick lookup in Clio

Clio divides its storage space into packets of N segments. A bitmap of N bits is stored for every N segments and for each logical log. A bit of this bitmap indicates whether the corresponding segment contains or not a record of the logical log (see Figure 13).

The scheme is recursive. At the second level, for each logical log, a bitmap of N bits is kept for every N^2 segments. A bit of this bitmap indicates whether the N corresponding segments contains or not a record of the logical log, i.e. if the bitmap of level one is not null.

These meta-data are structured as a search tree permitting to find the whole set of segments that contain at least one record of a given logical log. They are stored in a particular logical log of which records are located on regular and well-known locations. Consequently, additional meta-data are not needed to find the records of this particular logical log.

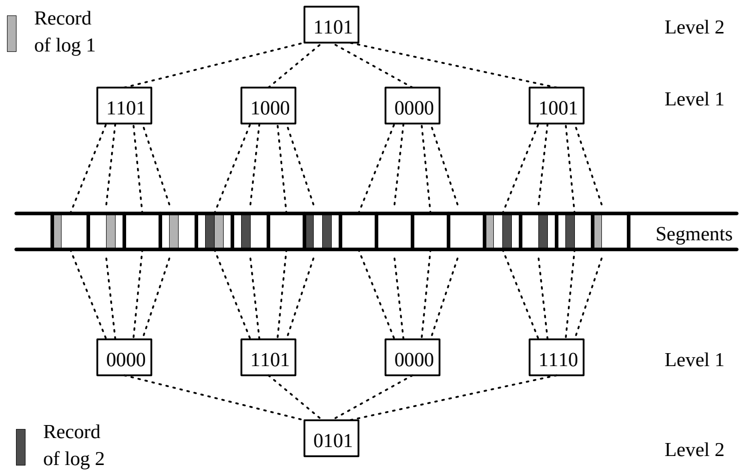

Figure 13: Bitmaps of record presence

For each logical log a hierarchy of bitmaps indicates if log records are present or not in physical log segments. In this example, the interval N is 4 (as in Clio implementation). For each logical log, a bitmap of 4 bits of level one indicates for every 4 segments if some record of the logical log are present in the 4 segments. Every 16 segments, a bitmap of level two indicates if the 4 last bitmaps of level one are null or not.

6.2 Quick lookup in Camelot

Camelot uses a modified version of the Clio method. The highest `LSN` corresponding to the record stored in the segments is stored with the bitmap.

These data are stored at any position of the physical log. Records corresponding to these data contain a backward pointer to the previous record of this logical log. The last record is saved in a checkpoint, or can be found by reading backward the last new record written on the physical log.

6.3 Quick lookup in KITLOG

The KITLOG solution is a simplification of these methods. For each logical log the identifier and location of the first record is kept in memory. For each logical log, the identifier and location of the last record is added to these meta-data, at regular intervals. Starting with a given record identifier, a pair of registered record identifiers can be found determining an interval on the physical log where the record is stored.

These meta-data are cached in memory and saved in a particular logical log. Similar meta-data are kept on this logical log but are only kept in memory. If these meta-data are lost they can be reconstituted lazily when needed by reading the physical log.

The interval of time between the registration of record location and identifier is customisable for each logical log permitting to balance the space overhead due to the storage of these data and the time overhead for record lookup.

Unlike Clio's or Camelot's methods, the meta-data volume is neither fixed, nor proportional to the number of records.

7 Summary and conclusions

Logging techniques are the way to manage the append-only semantic that presents two main characteristics: it is particularly adapted to support fault-tolerant applications and provides very good write performances.

Append-only storage techniques for general-purpose logging systems must be designed following specific constraints: avoiding fixed-position meta-data, keeping the write speed advantage inherent to the append-only semantics and being fault-tolerant. As a support for fault-tolerance, the logging techniques must also be fault-tolerant and do not use fault-tolerant mechanisms relying on logging. Consequently non-traditional solutions have to be found to fulfill this goal.

The state of the art for logging technique presents a set of convergent features and shows that research has still to be done in this domain:

- Segmentation of the storage medium is a common characteristic of all the systems (see §3.5). It is used to apply the different techniques on small independent parts of the physical medium providing fault-tolerance, possibilities of parallelisation between logging and compaction, and random access to data.

- Compaction techniques need some improvements to solve the problem of automatic garbage-record detection for general-purpose systems (see §4.4.1). Investigations have to be done to compact the medium by parts without loosing the record storage order as it happens with Sprite LFS (see §4.3.3).

- The problem of finding the end of the physical log shows the trade-off between an exact solution (avoiding to loss some work) and having good write performance (see 5.4).

- Finally, the different algorithms for quick data lookup all present a trade-off between space overhead and efficiency of read access (see §6).

Research in general-purpose logging systems is important because the need for logging is increasing in number and variety every day [Daniels 1990, Satyanarayanan et al 1992a, Satyanarayanan et al 1992b, Ruffin 1994]. General-purpose systems present new challenges; for instance, a general-purpose system must be customisable to its different client needs [Ruffin 1993]. The lack of semantics knowledge, partially solved by logical logs, is still a problem for automatic garbage-record detection. Some work has been done to bring more semantics at the logging system level by typing records [Wirz and Nett 1993] but more general solutions have to be investigated.

Acknowledgements

I would like to thank Marc Shapiro for advising my thesis (from which this paper is adapted). I thank the other members of my thesis committee: Claude Girault, Sacha Krakowiak, Patrick Valduriez, Willy Zwaenepoel, Hank Levy and Marc Guillemont for encouraging me to publish this paper. I thank Ross Finlayson and John Ousterhout for their answers to my questions on Clio and Sprite LFS. Thanks also to the first readers of this paper for their helpful comments: Laurent Amsaleg, Laurent Daynès, Bertil Folliot, Miguel Mira da Silva, Marc Shapiro and Dimitri Tombroff.

References

Baker, H. 1978. List Processing in Real Time on a Serial Computer. *Comm. of the ACM*, Vol. 21, N°4, pp. 280–294. Apr.

Cheriton, D. 1987. UIO: A Uniform I/O Interface for Distributed Systems. *ACM Trans. on Comp. Syst.*, Vol. 5, N°1, pp. 12–46. Feb.

Cohen, J., and Nicolau, A. 1983. Comparison of Compacting Algorithms for Garbage Collection. *ACM Trans. on Prog. Lang. and Syst.*, Vol. 5, N°4, pp. 532–553. Oct.

Collins, G. 1960. A Method for Overlapping and Erasure of Lists. *Comm. of the ACM*, Vol. 2, N°12, pp. 655–657. Dec.

Daniels, D. S. 1988. Distributed Logging for Transaction Processing. PhD Thesis, Carnegie-Mellon Univ., Pittsburgh, PA (USA), Tech. Rep. CMU-CS-89-114. Dec.

Daniels, D., Haskin, R., Reinke, J., Sawdon, W. 1990. Shared Logging Services for Fault Tolerant Distributed Computing. *Proc. of the fourth ACM SIGOPS Eur. Work.*, Bologna (Italy). 3–5 Sep. 1990. Oper. Syst. Review, Vol. 25, N°1, pp. 65-68. Jan. 1991.

Daniels, D. S., Spector, A. Z., and Thompson, D. S. 1987. Distributed Logging for Transaction Processing. *Proc. of the* ACM SIGMOD Int. Conf., San Francisco, CA (USA). 27-29

May. SIGMOD Record, Vol. 16, N°3, pp. 82–96. Dec.

DIJKSTRA, E., LAMPORT, L., MARTIN, A., SCHLOTEN, C., AND STEFFENS, E. 1978. On-the-fly Garbage Collection: an Exercise in Cooperation. *Comm. of the ACM*, Vol. 21, N°11, pp. 966–975. Nov.

DOUGLIS, F., AND OUSTERHOUT, J. 1989. Log-Structured File Systems. *Intellectual leverage, COMPCON Spring 89, Thirty-fourth IEEE Computer Society Int. Conf.*, San Francisco, CA (USA). 27 Feb – 3 Mar.

FENICHEL, R., AND YOCHELSON, J. 1969. A LISP Garbage Collector for Virtual Memory Computer Systems. *Comm. of the ACM*, Vol. 12, N°11, pp. 611–612. Nov.

FINLAYSON, R. S. 1989. A Log File Service Exploiting Write-once Storage. PhD Thesis, Stanford Univ., Stanford, CA (USA), Tech. Rep. STAN-CS-89-1272. Jul.

FINLAYSON, R. S., AND CHERITON, D. R. 1987. Log Files: An Extended File Service Write-Once Storage. *Proc. of the Eleventh ACM Symp. on Oper. Syst. Principles*, Austin, TX (USA). Oper. Syst. Review, Vol. 21, N°5, pp. 139–148. Nov.

HAGMANN, R. 1987. Reimplementing the Cedar File System Using Logging and Group Commit. *Proc. of the Eleventh ACM Symp. on Oper. Syst. Principles*, Austin, TX (USA). Oper. Syst. Review, Vol. 21, N°5, pp. 155–162. 8–11 Nov.

HAGMANN, R. B., AND GARCIA-MOLINA, H. 1991. Implementing Long Lived Transactions Using Log Record Forwarding. Computer Science Laboratory, Xerox, Palo Alto, CA (USA), Tech. Rep.-CSL-91-2. Feb.

HASKIN, R., MALACHI, Y., SAWDON, W., AND CHAN, G. 1987. Recovery Management in QuickSilver. *Proc. of the Eleventh ACM Symp. on Oper. Syst. Principles*, Austin, TX (USA). Oper. Syst. Review, Vol. 21, N°5, pp. 75–86. 8-11 Nov.

KAUNITZ, J., VAN EKERT, L. 1984. Audit Trail Compaction for Database Recovery. *Comm. of the ACM*, Vol. 27, N°7, pp. 678–683. Jul.

KEEN, J. S., AND DALLY, W. J. 1993. Performance Evaluation of Ephemeral Logging. *Proc. of the* ACM SIGMOD Int. Conf., Washington, DC (USA). 26–28 May. SIGMOD record, Vol. 22, N°2, pp. 187–196. Jun.

LIEBERMANN, H., AND HEWITT, C. 1983. A Real Time Garbage Collector Based on the Lifetimes of Objects. *Comm. of the ACM*, Vol. 26, N°6, pp. 419–429. Jun.

MCBETH, J. 1963. On the Reference Counter Method. *Comm. of the ACM*, Vol. 6, N°9, pp. –575. Sep.

MCCARTHY, J. 1960. Recursive Functions of Symbolic Expressions and their Computation by Machine. *Comm. of the ACM*, Vol. 3, N°4, pp. 184–195. Apr.

ROSENBLUM, M., AND OUSTERHOUT, J. K. 1991. Design and Implementation of a Log-structured File System. *Proc. of the Thirteenth ACM Symp. on Oper. Syst. Principles*, Pacific Grove, CA (USA). Oper. Syst. Review, Vol. 25, N°5, pp. 1–15. 13–16 Oct.

RUFFIN, M. 1992A. KITLOG: Un service de journalisation générique. Thèse de Doctorat de l'Université Pierre et Marie Curie – Paris VI (France). INRIA (France), Thèse TU-0205. 14 Sep.

RUFFIN, M. 1992B. KITLOG: a Generic Logging Service. *Proc. of the Eleventh Symp. on Reliable Dist. Syst.*, Houston, TX (USA), pp. 139–146. 5–7 Oct.

RUFFIN, M. 1993. An Architecture for Fault Tolerant Storage Systems. *Fifth Eur. Work. on Dependable Comp.*, Lisbon (Portugal), 17–19 Feb.

RUFFIN, M. 1994. A Survey of Logging Uses. INRIA (France), Tech. Rep. BROADCAST-36. Also available as Univ. of Glasgow (Scotland), Tech. Rep. FIDE$_2$-94-82. Feb.

SATYANARAYANAN, M., KISTLER, J. J., KUMAR, P., AND MASHBURN, H. 1992A. On the Ubiquity of Logging in Distributed File Systems. *Proc. of the Third Work. on Workstation Oper. Syst.*, Key Biscayne, FL (USA), 23–24 Apr.

SATYANARAYANAN, M., STEERE, D. C., KUDO, M., AND MASHBURN, H. 1992B. Transparent Logging as a Technique for Debugging Complex Distributed Systems. *Proc. of the Fifth ACM SIGOPS Work. on Models and Paradigms for Dist. Syst. Structuring, Mont Saint-Michel (France)*. 21–23 Sep.

SPECTOR, A. Z. 1987. Distributed Transaction Processing and the Camelot System. Carnegie-Mellon Univ., Pittsburgh, PA (USA), Tech. Rep. CMU-CS-87-100. Jan.

UNGAR, D. 1984. Generation Scavenging: a Non-disruptive High Performance Storage Reclamation Algorithm. *Proc. of the ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on Practical Soft. Development Env.* Apr. ACM SIGPLAN Notices, Vol. 19, N°5, pp. 157–167. May.

WIRZ, B., AND NETT, E. 1993. A Generic Log-Service Supporting Fast Recovery in Distributed Fault-Tolerant Systems. *Proc. of the IEEE Work. on Advances in Paral. and Dist. Syst.*, Princeton, NJ (USA). 6 Oct.