KITLOG : a generic logging service

Michel Ruffin

KITLOG

A Generic Logging Service

Un service de journalisation générique

Michel RUFFIN

Projet SOR
Institut National de Recherche en Informatique et en Automatique
&
Laboratoire de Méthodologie et Architecture des Systèmes Informatiques,
Université Pierre et Marie Curie, Paris VI

———————

Tel: +33 (1) 39 63 54 26, E-mail: ruffin@sor.inria.fr

May 1992

Rapport de Recherche INRIA N°1678

Abstract

Many applications use a log to store state, either to "redo" past actions, or to correct possible faults, errors or loss of consistency. A generic logging service should cater to the variable and even antagonistic needs of applications, without imposing overhead on applications which do not use all its functions. For instance, a transaction manager needs a different degree of reliability than a debugger.

$K_I{}^T L^O{}_G$ *provides an original solution to this problem by decomposing and encapsulating logging functions. Five main mechanisms are distinguished: buffering policy, distribution of records, replication of records, sharing and multiplexing mechanisms, and management of physical media. Each mechanism is embodied in an object class. For each class, multiple policy implementations can be provided. Instances of these classes are stackable in any appropriate number or order. An application programmer customizes his logical log to a set of failure assumptions by selecting adequate classes, instantiating them, and connecting the instances together. Thus, the application pays only the cost of the logging functions it chooses.*

Keywords: Log management, generic service, object composition, $K_I{}^T L^O{}_G$.

Résumé

De nombreuses applications utilisent un journal pour stocker des états, soit pour refaire des actions passées, soit pour corriger les effets de pannes, d'erreurs ou de pertes de cohérence dues au parallélisme. Un service générique de journalisation doit s'adapter aux besoins variables, voir antogonistes, des applications, sans imposer un surcoût aux applications qui n'utilisent pas toutes ses fonctions. Par exemple, un gestionnaire de transactions nécessite un degré de fiabilité différent d'un débogueur.

$K_I{}^T L^O{}_G$ *fournit une solution originale à ce problème en décomposant et en encapsulant les différentes fonctions de la journalisation. Cinq principaux mécanismes sont distingués : la politique de gestion des tampons, la distribution d'enregistrements, leur réplication, les mécanismes de partage et de multiplexage, et la gestion des ressources physiques. Chaque mécanisme est représenté par une classe d'objets. Pour chaque classe, différentes réalisations peuvent être fournies. Les instances de ces classes sont composables dans n'importe quel ordre approprié et en n'importe quel nombre. Un programmeur d'applications adapte son journal logique à un ensemble d'hypothèses de pannes, en choisissant les classes adéquates, en les instanciant, et en connectant les instances ensemble. En conséquence, l'application paye uniquement le coût des fonctions de journalisations qu'elle utilise.*

Mots clefs : Journalisation, service générique, composition d'objets, $K_I{}^T L^O{}_G$.

I Introduction

Two main observations have motivated this work. The first one is that distributed operating systems provide different reliability levels. Most of the traditional algorithms that ensure fault tolerance (such as two step commitment [Baer et al. 81], recovery protocols [Gray et al. 81] and concurrency control [Bernstein and Goodman 80]) need a reliable log. Logs are the greatest common divisor of most reliability mechanisms.

The second incentive stems from the increasing number of applications implementing logs customized to their specific needs. Until now, a generic logging tool satisfying all their requirements has not existed. Log management is hard to implement correctly and efficiently. A generic service would allow application programmers to save their time.

As examples, Camelot [Daniels et al. 87, Daniels 88] and QuickSilver [Haskin et al. 87] use a log for transactions management and failure recovery. Emacs uses an in-memory log to undo and redo file modifications. Instant Replay [Leblanc and Mellor-Crummey 87] uses a log to replay debugging sessions. Isis [Birman et al. 89] implements a log for communications reliability. Sprite [Rosenblum and Ousterhout 91] stores its complete file system on a log increasing its write efficiency.

This paper is composed of two parts. First, it describes a *generic service* for logging, a high-level object oriented design called the $K_I^T L^O{}_G$ model. Then, it presents one of the possible log management policy design, corresponding to the current prototype.

II Requirements

Designing a generic logging service starts with a study of the application requirements. These have points in common and divergent features.

Commonalities

The common characteristic of all logs is the append-only storage semantics. In this section, we describe these semantics, illustrating them with an interface (see Figure 1). This interface is a basic log one. Operations names and arguments can be changed from one implementation to another but the operation semantics must be unchanged.

For the sake of clarity, we supposed that a log storage unit exists called the record, and constituted of an opaque data block of any size. Because of the append-only semantics, records are stored in the order they arrive. We also assume that records have a record identifier (`Rid`) that is unique within the log. Values of different `Rid` increase with time, reflecting the record arrival order.

Appending records
`put (Lid, Rid, D, S)`
`put (Lid, &Rid, D, S)`
Giving obsolete record list to the system for compaction
`invalidate_records (...)`
Reading records backward
`get_last (Lid, &Rid, &D, &S)`
`get_prev (Lid, &Rid, &D, &S)`
Reading records forward
`get (Lid, &Rid, &D, &S)`
`get_next (Lid, &Rid, &D, &S)`
Forcing writes on storage medium
`flush ()`

Figure 1: Basic log interface

`Lid`: log identifier, `Rid`: record identifier, `D`: Opaque record data block, `S`: record size, `&`: reference. With the first `put()`, applications choose the `Rid` value. With the second one, the logging service generates a `Rid`. `Invalidate_records()` supports different designations of obsolete records. `Get()` with `Rid` of zero returns the first record of the log.

Write accesses appends records to the end of the log with the `put()` operation. A log record may become obsolete: for instance, the record life time of an aborted transaction is that of the transaction. Useless records can be compacted out of the log whenever possible. The `invalidate_records()` operation allows clients to indicates obsolete records to the logging service. It can have mutliple forms depending on the indication method (developed in §V.5).

For most applications, reads are uncommon compared to writes. Reads are generally needed when a problem occurs. Reads may be backward or forward. A backward read starts at the end of the log (`get_last()` operation) and are done in the reverse order of storage (`get_prev()` operation). Backward reads support undoing past actions. Forward reads supports redo of past actions. By restoring a global state (or checkpoint) (with `get()`) and reading records corresponding to further states modifications (with `get_next()`), applications are able to recover all the different execution steps they need.

Finally, a `flush()` operation forces data writes onto the storage media. When this operation returns, all the data buffered in memory are guaran-

teed to be written on their final storage medium.

$K_I{}^T L^O{}_G$ supports *logical logs*. Logical logs are client-level logs, while *physical logs* are a set of records from different logical logs multiplexed on a single storage medium (for the sake of clarity, we suppose that there is a single physical log by storage medium). For applications,[1] a log is represented by an unique log identifier (`Lid`). An application may use multiple logs. A log may be shared, or not, between different processes of a particular application, or between separate applications. Records from different logs can be multiplexed, or not, on a storage medium.

Differences

Logs present several different features. The differences may be quantitative, e.g. different level of reliability, or may be antagonistic qualities, e.g. availability versus consistency.

Depending of the application, the log provides a certain *degree of reliability*. This includes the types of faults which can be tolerated (e.g. site, disk or network), as well as the number of simultaneous occurrences of the same fault type. For instance, some application might require resistance against 5 simultaneous disk faults.

A log's response to some failure should also be tailored to its client's needs. For instance, an application replicating data with strong consistency semantics might impose strong consistency on log replica, whereas an application needing a high availability degree can choose a weak consistency protocol for its log replication.

Records have variable lifetimes. A log can be composed of both persistent and transient records. For example, to ensure the transaction persistence property, old persistent record must be archived while transient records (for aborted transactions) have to be compacted out of the log.

Finally, some applications require network transparency, whereas others need to control the location (site, disk) where the log is stored.

[1] We use an extended definition of the word application. Applications can be of high or low level (e.g. a financial software using audit trails, Emacs or a recovery manager). For the logging service (implemented as a set of logging servers), application are clients that communicate via a library.

III General architecture

III.1 Basic principles

The $K_I{}^T L^O{}_G$ model breaks down logging mechanisms into elementary logging functions like Ficus for file systems [Heidemann and Popek 91], x-Kernel for communications [Hutchinson and Peterson 88] or System V for streams [Ritchie 84].

Each function is encapsulated in a distinct object class, called a *building block class* or a *block class*. All block classes have the same interface, since they all implement the same log semantics. In object terms, all building block classes conform to a single *block type*. An application programer *composes* the block classes and instantiates them into a directed acyclic graph of objects. A process uses a particular entry point in the graph which defines the particular logical log properties of the process, depending on the different building blocks that can be reached by this entry point.

In the remainder of this paper, a *successor* of an object is either one of the object's direct successors in the graph, or by transitivity one of the successors of its successors. We define in the same way a *predecessor* by replacing in the previous definition the term *successor* by *predecessor*.

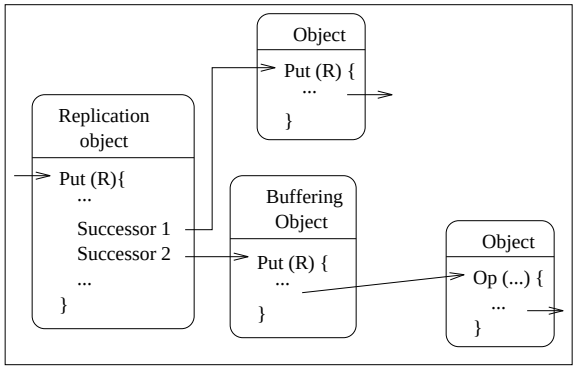

Figure 2: Object composition

Block operations[2] (like `put()`, `get()`, etc.) in *block object composition* behave in the following way: operations and arguments move from a graph head towards the final storage media (graph leaves) following the edges of the graph. When a record (or a set of records) crosses a block object, the function embodied by its class is applied to the record. This comportment is similar to that of System V streams operations.

[2] These operations include the operation of the previous described interface with some modifications and new private operations for inter-blocks invocations. The prototype block interface is describe in §V.2.

For instance, in Figure 2, a `put()` of record R on a replication object generates a replicated invocation of its successors `put()` operation. While, a `put()` of record R on buffering object caches the record in memory. If the buffer is full, the `put()` operation flushes the buffer by invoking some appropriate operation `Op()` of its successor that allows it to transfer multiple records to its successor in a single operation. This operation will continue to propagate down the graph until it meets a buffer or a block interfacing a medium which is able to store data.

Object composition is based on the separation of type and class. The composition is feasible because, the successor to any block class conforms to the same block type interface.

Two kind of objects play a special rôle as head and tail of a log composition graph. A non privileged client's entry interface to its graph is a *logging view object*. The final exit interface from the graph onto a storage medium is a *medium object*. While logging views belong to a separate type (their interface is roughly that of Figure 1), medium objects are instance of the medium block class which conform to the block type.

In the current implementation, using a mini-description language, a graph is described by two configuration files: one file for the server parts and one for each client address space part. Graph parts shared by different applications are protected by being in the server address space. We suppose that graphes are correctly build. Before servers create or reconfigurate the system part graph, instantiating and connecting objects together, it verifies very few things (no cycle, no unacceptable configuration, etc.). Tools are provided to help graph building.

III.2 Elementary block classes

Log functions have been broken down into five elementary classes: buffering, medium, sharing, distribution and replication.

Buffering

A buffering class implements log semantics in memory. It may have multiple rôles. The simplest is caching inputs and outputs. It may also act as filter. Cached records are temporary or permanent. If the client application has declared a list of obsolete records, they are compacted out of the buffer, before transferring records to the buffering block successor.

Other rôles are possible, such as data storage structuring. For example, it is possible to gather records of some particular kind contiguously, rather than in arrival order, by using a different buffering block object for each kind, and multiplexing the record packets with a sharing block object. If needed the flushing level (number of flushed successors) can be controled by clients applications.

Medium

Medium classes are the interface to resources which do not necessary conform to the block interface. Each medium sub-class drives a particular kind of resource, e.g. disk, tape, terminal, process or file. A medium object has no successor within the log graph.

Sharing

A sharing block class has multiple predecessors. It implements a sharing policy of its successor by its predecessors. It manages concurrent access, record multiplexing and access control. This control is perfomed at connexion time, i.e when connecting an object as a predecessor of a sharing object.

Distribution

A distribution class encapsulates distribution and communication between building blocks on different sites and address spaces. It establishes communication, and sends and receives records. Such a class encapsulates communication protocols and policies chosen in case of communication failures.

This object class does not cache records. If one wants to gather records within a single message, a distribution object must have a buffering object as predecessor. When a buffer flush is necessary, the buffering object invokes the distribution object operation that transfers record packets between two blocks, initiating the emission, in a single message, of the buffer of records.

Replication

Finally, a replication class manages record replication on different successors. A replication class embodies a replication policy. It implements a recovery algorithm that depends on the replication protocol and the synchronization of operations on its successors. A replication block handles a fixed number of replica. This number is called its *replication degree*. Depending of the way successors are bound to the block (see §III.4), their number can be greater or equal than the block replication degree and the replica can be stored by different subsets of successors.

Implementation of different policies

A building block function may be implemented by several classes, each implementing a different protocol. For instance, various replication classes are possible, implementing strong, weak or release consistency.

Other classes

Client programmers can add new logging functions by adding new block classes. For example, an encryption class could enforce log security, or a class could multiplex client-level sub-logs into a single service-level log. These possibilities are open issues.

III.3 Log example

Figure 3 shows a complete example of a log instantiation. The log is replicated on two sites: site A is a storage site; site B is a control site. A log server on site A enables different clients to share the logging media via a sharing object. On site A, log records are cached by a buffering object before being replicated on two disks. These objects are instantiated in the server, making disk sharing possible.

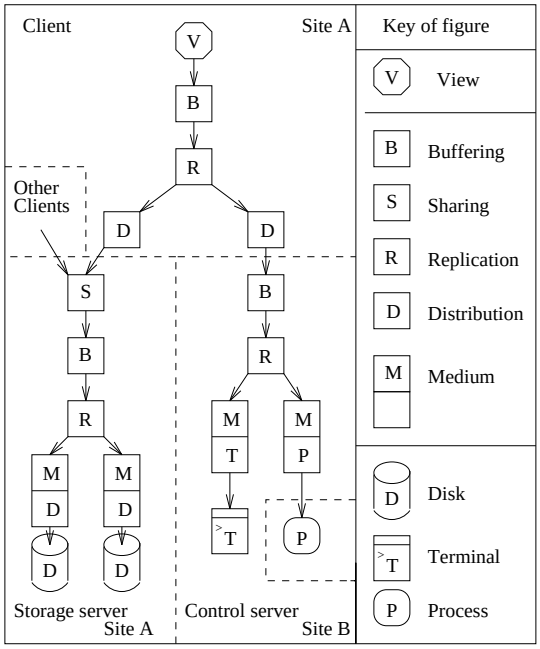

Figure 3: Log example

The second sités server buffers client records before displaying them on a terminal and transferring them to a monitoring process.

Starting at the top of the graph, the client logging view transfers records to the two servers via a replication object. Distribution objects are needed, first to communicate with the remote server B, and second, to change the address space on site A. The buffer object gathers records before sending them to the servers to minimize the number of messages.

III.4 Dynamic reconfiguration

The graph can be re-configurated dynamically to react to faults of system elements, depending on the way objects have been bound together. The binding method is defined at graph configuration time by the successor method designation.

Object successors can be designated in two different ways. First by identity: a client names sites and storage media on which the log will be stored. For instance, one can request something such as: "a log replicated on disks X and Y of site A and on disks X and Z of site B".[3] This designation method corresponds to static binding, which does not enable any reconfiguration.

The second designation method is functional: successors are designated by their class name rather by their identity. Thus, a client could request: "a log replicated on two sites, and on two disks of each site". In this case, the logging service chooses the sites and the disks among those available and reconfigures upon failure. Available resource knowledge comes from system graph part which describes the logging devices and their access mode (i.e. what part of graph must be interposed between an application object and the device). Failure detection is provided by the logging service. We will not develop further this point which is beyong the scope of this paper.

A block that can have different successors by dynamic reconfiguration must be able to write records on variable successors or to search for records among multiple possible successors. As this is typically a replication problem, we limit dynamic binding to the replication block class. If one does not need replication but wishes to use dynamic reconfiguration, a replication block with a replication degree of 1 can be used.

For example (see Figure 4), a client has specified, using functional designation, a log replicated on two sites and on two disks on each site. Site C has three available disks, and three sites have at least two disks. The system has chosen two disks of site B and two of site C. When a disk fails on

[3] Currently, device names are unix ones (e.g. /dev/rsd0c). Site names are standard machines ones. Objects are designated by a class name and a symbolic name arbitrary chosen by graph builders.

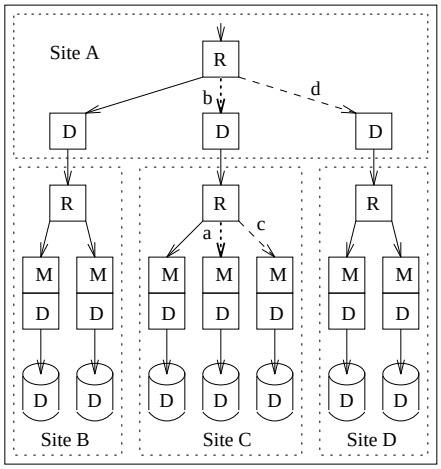

Figure 4: Automatic reconfiguration

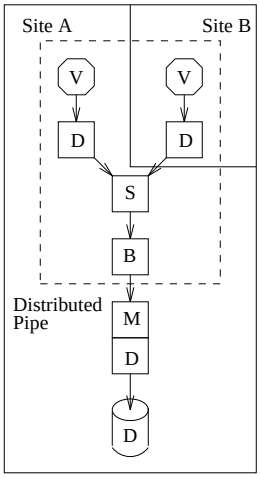

Figure 5: Reliable distributed pipe

site C (dotted arrow "a") an other one can be chosen by the system (dashed arrow "c"). Similarly, a failure of site C (dotted arrow "b") leads the logging service to elect site D (dashed arrow "d"). If the site C had only two available disks, a disk failure would lead to the election of another site. Depending on the replication object class chosen, when a failure occurs, different reaction are possible. For instance, reachable log records can be copied in order to maintain two full copies of the logical log.

IV Further examples

In order to illustrate the properties of the $K_I{}^T L^O{}_G$ model and its varied uses, we present two sample logging configurations. The first one illustrates some of the unusual possibilities allowed by the model. The second one simulates an existing logging tool.

IV.1 Reliable Distributed Pipe

As pipe semantics are very close to log semantics, the following example (see Figure 5) shows how one can build a log graph to implement a reliable distributed pipe.

A shared memory is obtained by multiplexing two logging views V on a buffering object B via a sharing block S. Distribution is provided by the intermediate distribution block D on site B. The D block on site A permits entering the site A server address space.

If one client uses the `put()` operation while an other one uses the `get()` operations, a distributed pipe is obtained. The reliability is due to the addition of a medium block and can be enhanced by replicating the log on different sites and disks.

IV.2 The Camelot Example

This second example shows how to approximate the Camelot logging facilities [Daniels 88] using the $K_I{}^T L^O{}_G$ services. Figure 6 shows the corresponding graph.

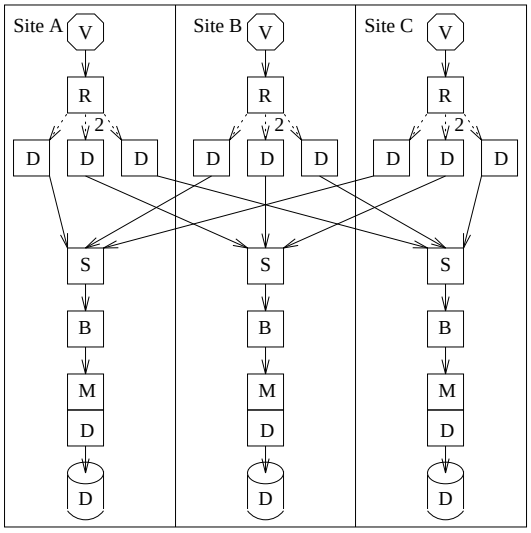

Figure 6: A Camelot-like log

Within Camelot, each record is replicated on N sites among the M sites that provide a logging service. In this example, N is 2 and M is 3. Three

client transaction managers, are logging on three different sites.

Clients, via a view object, pass records to a replication object R dynamically bound to 2 distribution objects. In Figure 6, dynamic binding is represented by dotted arrows; the replication degree (2) is indicated by a number on the edges. Records are sent to the appropriate site successor with a distribution object and multiplexed in a buffer object with a sharing object, before being written on a disk with a medium object.

If a site failure occurs, in Camelot, a new site is chosen to maintain the replica number. With $K_I{}^T L^O{}_G$, the system re-configurates the graph by binding a new successor to the replication object.

Obviously the storage organization must be customized to Camelot needs and the five main classes must be written to implement particular Camelot logging algorithms. For instance, the Camelot recovery protocol should be implemented by the replication class.

V Log management policy

Within the $K_I{}^T L^O{}_G$ model, multiple log management policies can be designed.

A log management policy is characterized by a particular storage organization and the semantics of the block operations. Depending on this storage organization, a set of meta-data types are defined to manage the physical logs. This is reflected by a particular block interface definition. For instance, in the current prototype, we have defined `Rids` and `Lids` as specific structures. They are gathered (and stored) in a single other structure: the `Trailer`, appearing in most of the operation interfaces. Consequently for a given storage organization, a particular block interface is defined, and is characterized by a *block abstract type*.

$K_I{}^T L^O{}_G$ block abstract types are similar to the abstract interfaces of Lipto [Druschel et al. 91] and their definition can be adapted from that of Emerald abstract types [Black et al. 87]: *A block abstract type defines the interface of a block object–the set of operations supported, their signatures and (in principle) their semantics. An operation signature includes the operation name and the types of the arguments and results.* Consequently, no implementation is tied to a block abstract type. A block class is one of the possible implementation of the block abstract type.

A single management policy does not meet all application requirements. Consequently, multiple block abstract types and sets of associated classes must be designed.

This section describes the $K_I{}^T L^O{}_G$ prototype log management policy: storage organization, block abstract type interface and semantics, and some algorithms: finding the log end, finding records, and compaction.

V.1 Storage organization

Whatever the storage medium, e.g. memory, message, disk, the storage organization is the same. The storage medium is decomposed into fixed-size chunks or *sectors*. The sector length is a multiple of the physical disk sector length,[4] the unit for writing or reading. The sector is an exchange unit between objects in the graph, avoiding unnecessary copies when transferring data to each other. For data exchange, a block can pass to its successor or predecessor a pointer referencing the beginning of a sector in a buffer. This buffer belongs to the block or to one of its successors or predecessors. When moving through the graph, data must be copied in the following cases: when arriving in a buffering block (if these blocks do not manage a set of pointers on record contents), when changing site or address space, and when being written on (or read from) a physical device.

Records have two parts: the opaque data block (`R`) and a descriptor, called a *Trailer* (`T`). A trailer is composed of a log identifier (`Lid`), a record identifier (`Rid`) and the size of the opaque data block (`S`). When writing a record, the opaque data block is appended. Then, the trailer is written at the end of the last sector enclosing a part of the record body. If there is not enough space within the sector, the trailer is written in the next one.

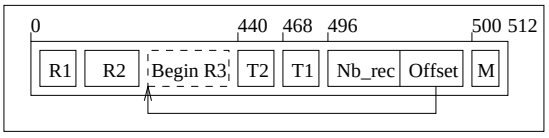

Figure 7: Sector organization

`T`: record trailer, `R`: record opaque data block, `Nb_rec`: number of record trailers in the sector, `Offset`: number of bytes from the begining of the sector to the end of the last record ending in the sector. Dashed records are records split among several sectors.

Every sector finishes with a marker, `M` (see Fig-

[4] The sector length is 512 bytes in the current implementation. We assume that an appropriate length can be found for all kinds of devices.

ure 7). Markers for contiguous sectors are strictly increasing. They allow the logging service to find the physical log end on a medium at boot time, and to restore consistency after a failure during a medium compacting (see §V.3). Each sector also includes the number of trailers in the sector (`Nb_rec`) and the offset corresponding to the first byte after the last end of opaque data block in the sector (`Offset`).

These data are sufficient to read the log forward or backward, given a random sector of a physical log. For instance, Figure 8 shows an example of record storage. Given the Rid and Lid values of record Rd, its contents is found by reading sectors 2 and 1. In sector 2, the number of trailers indicates two valid trailers. The size of each opaque data block ending in the sector is found by examination of the trailers. The Rd data block end is at the offset `Offset` (represented by the arrow) minus the size of Re data block minus one byte. Rd's beginning is obtained by subtracting its size (modulo the sector length). Finding the sectors 1 and 2 is another problem discussed in §V.4.

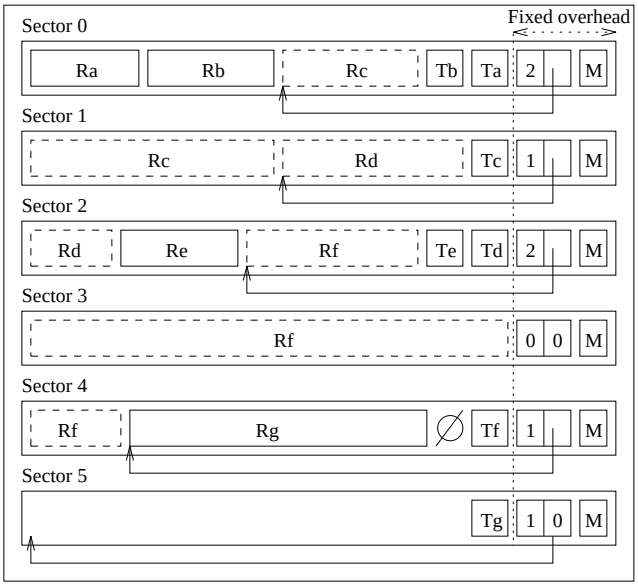

Figure 8: Storage example

After any partial medium failure, this organization enables the recovery of records on undamaged parts of the medium because there is no fixed-placed meta-data.

The space overhead is low. In the current implementation, the fixed storage overhead per sector is 20 bytes (the sizes of marker `M`, `Offset` and `Nb_rec`), while the overhead per record (the size of a trailer) is 28 bytes. Fixed overhead can be reduced by increasing sector length, but the greater the sector length is, the greater is the probability of losing space in incomplete sectors.

V.2 Block interface

Figure 1 describes the interface seen by a client application (e.g. the logging view interface), while Figure 9 gives the most interesting parts of the building block abstract type interface. Logging view interfaces operate on logical logs while block ones manage physical logs.

The symbol `&` is a reference passing operator. In the "get operations" (`get()`, `get_last()`, etc.), the `Lid` field of the `Trailer` structure is set by the caller, whereas the `Rid` field is only set to call the `get()` operation itself and is a return value for other kinds of gets. `Contents` and `Buffer` are passed by reference, because their size is only known to the caller. These interface methods can only be called by a block object belonging to the same block abstract type, or by a logging view object.

`Attach()` and `detach()` bind and un-bind a block object successor to the current object, respectively.

`Put()` is used to append a single record into a write buffer of a buffering block. If the current block is not a buffering one, the `put()` operation is transmitted to its successor(s).

```
attach (&Block)
detach (&Block)
-----------------------------------------
put (Trailer, &Contents)
get (&Trailer, &Contents, Max_size)
get_last (&Trailer)
get_prev (&Trailer)
get_first (&Trailer)
get_next (&Trailer)
flush (Level)
prefetch (Lid, Rid)
invalidate_records (...)
infos (&Infos)
-----------------------------------------
multiple_put (Nb_sect, &Buffer)
multiple_get (Lid, Rid, Nb_sect,
                    &Buffer, Max_size)
```

Figure 9: Block interface

`Get()` returns the contents of a record whereas `get_last()`, `get_prev()`, `get_first()` and `get_next()` methods only search record identifiers on medium, buffer, tables, etc. (depending on the block nature and implementation). This

separation between get operations reflects the difference between looking records up and getting their contents.

`Flush()` copies the contents of the current block write buffer (if there is one) into the first successor(s) that is (are) able to store data and call the flush operation of its successor(s). If the `level` argument is set to -1, a recursive flush on all the object successors is executed. Set to another value, the flush is limited to the equivalent level in the graph. For instance, a flush of level two will successively call a flush on the current object, on its successors and on all the successors of its successors. This allows clients to control the flush level when they use multiple buffers for structuring data (see the buffering block description in §III.2).

`Prefetch()` is the converse operation. It asks its successors to load into their read buffer a "log data segment" enclosing a given record. This operation prepares future get operations.

`Invalidate_records()` gives the logging service a list of obsolete records that can be compacted out of the log. Multiple record description forms are possible (explained in §V.5).

`Infos()` returns some information about the object: its class, the list of its inherited classes, the object state, its successors list, etc.

Finally, `multiple_put()` and `multiple_get()` operations pass multiple sectors at once. The former passes `Nb_sect` sectors to its successors, whereas the latter asks successors for `Nb_sect` contiguous sectors including a particular record. This chunk of data must be put in the buffer `Buffer`, whose length is `Max_size`. These operations manage the copies of data chunks between building block objects.

V.3 Finding the log end

Finding the end of a physical log is a current problem that log designers encounter. The problem stems from the fact that due to physical log reliability, information on the log must be sufficient to recognize log data. The log service must be able to identify beginnings and ends of records, to find the log end, to determine if a fault occurs during a compaction, etc.

The flush operation when returns, guarentees that every record written on the medium can be recovered. Consequently, once a write is done, the logging service must be able to find the new log end without any in-memory information. But, log end location cannot be written at a fixed position of the storage medium. Such an approach would involve two writes on different medium places for each write of data on the physical log.

The traditional solution to this problem is to fill the log free space with zeros and to find with a binary search the last non-zeroed sector. This solution's drawback is the time cost, after a compaction or at medium initialization, needed to write zeros. Moreover, large records filled with zero complicate the searching algorithm.

Our solution is to use the sector marker to avoid filling free sectors with zeros. At regular intervals (say, every 5 minutes), the marker of the last sector of the physical log and its location are stored at the medium beginning. At boot time, the N first sectors of the log are read. If the device has been initialized, these N sectors have valid markers (marker validity is explained later) and they contain the location (s_l) and marker (m_l) of the last known log end. Then, a binary search looks up the last N contiguous sectors having increasing valid markers between the last known log end location and the medium end.

Markers are composed of: their generation date, a site identifier, and a compaction counter. The site identifier is that of the device site. During a compaction, a set of records from different sectors can be compacted in a single sector. The new sector marker is the greatest of the markers of the compacted sectors with an incremented compaction counter. A valid marker contains the local site identifier and the current compaction counter. The current compaction counter is incremented at the end of the compaction after having written the new log end location at the medium beginning). The probability of error in finding the log end depends on the value of N and on data representation (integer length, etc.). In the current implementation, N is 4 and the probability of error is very low (range 10^{-50}).

At boot time, the medium can be in three different states (see Figure 10). If the log has never been compacted, it is divided into two areas. The first one has valid increasing markers. The second one contains unspecified data. Finding the log end just requires being able to detect invalid markers

If the medium has been compacted several times, values of markers are increasing and decreasing following a sawtooth-shaped curve. As in the previous case, finding the log end is easy with the knowledge of the current compaction counter value.

The last case happens when a fault has occurred during a compaction. Before a compaction, additional information is written at the medium beginning: "entering a compaction". With this information, we know that a compaction has been

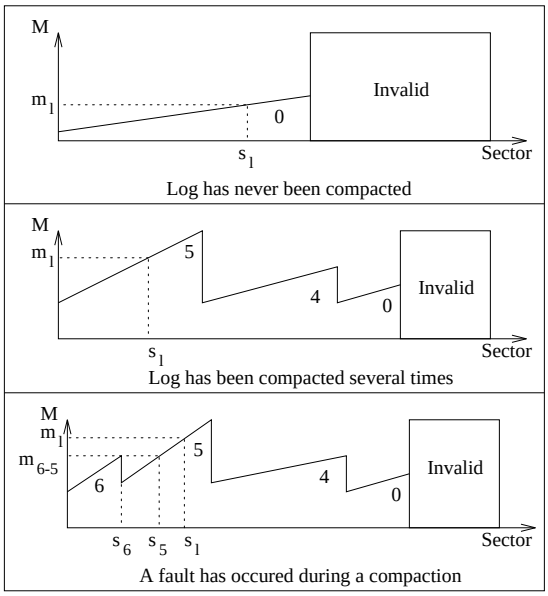

Figure 10: Marker values on the medium

Numbers on the Figure are the compaction counter values. m_l and s_l: marker and sector number of the last checkpointed log end. s_5: end of the compaction zone, beginning of sectors that have still to be compacted out. s_6: end of the already compacted sectors. m_{6-5}: the corresponding pair of markers (m_6, m_5), they differ by their compaction counter.

interrupted. Consequently, the medium beginning may contain markers which are not yet valid. Once the log end found, we search the beginning and the end of the compaction zone (between s_6 and s_5). s_6 can easily be found with a binary search between the medium beginning and s_l. Given the marker m_6, m_5 is found by decrementating its compaction counter. Finally a binary search between s_6 and s_l finds s_5. Once these zone limits are known, compaction continues.

If the first medium sectors, containing medium state and m_l and s_l values, are lost (we assume that this kind of event can be detected), they can be reconstituted by reading the two first saw-teeth.

V.4 Finding records

Within the above design description, a particular record look-up can only be done by reading a physical log from the beginning or the end in its entirety. In order to decrease the record search time, meta-data are maintained.

For each logical log, the first record `Rid` and its location on the storage media are registered and, regularly, the last record Rid and its location are added to these data. As `Rid`s increase with time, given a particular `Rid`, enables the logging service to find locations that define the limits of the records location.

These data are regularly stored on a particular log: the meta-data log. The similar meta-data on this log are kept in a traditional file system. Because of the unreliability of their storage, they are used only as hints to find record sectors. In case of loss, by reading the log storage media directly, we are able to reconstitute this information.

We have to study the interval of time for saving these data in order to balance the storage overload they produce and the time for searching records. More elaborate methods exist to accelerate records look-up [Finlayson and Cheriton 87] but this one has the advantage of simplicity.

V.5 Compaction

A list of obsolete records is necessary for physical log compacting. This list, produced by applications, is stored in the meta-data log, until compaction time occurs. Three methods have been selected to indicate these records to the logging service:

- `Lid, Rid:` Records are declared one by one. This is the most flexible and fine grain method.
- `Lid:` Delete a complete logical log.
- Delete all records with Rids lesser or equal than some value. This is useful for applications that can discard records older than some particular point (e.g. checkpoint).

Other designation methods are planed. For instance, it could be usefull for nested transactions to group records of a set of subtransactions in order to invalidate all these records when the top-level transaction is aborted.

Before compaction, a list of obsolete records must be prepared. Multiple choices can be done: having a partial or complete list, for instance. We suppose in the remainder that the list is complete and sorted by record location in memory.

A log compaction problem occurs when the compaction zone is not large enough to allow moving data by first coping it into a new place and then discarding the old copy. This method is necessary for avoiding data loss when a fault occurs during compaction. Compaction is done with the following algorithm:

- ❒ Write the new medium state: "entering compaction" at medium beginning.
- ❒ Search the first sector that encloses a record to be removed.
- ❒ Read it and some of the following sectors to fill a buffer.
- ❒ Compact the buffer: The marker of a new sector is the greatest marker among those of the compacted record sectors. The marker compaction counter is incremented.
- ❒ If there is enough space to write the buffer without corrupting valid data, write the new full sectors of the buffer in the compaction zone.
- ❒ Else,
  - Write them at the end of the log as a record of the *compaction log*.
  - Write the buffer full new sectors in the compaction zone.
  - Invalidate the compaction record.
- ❒ At compaction end, write the new physical log end information at the medium beginning and then, remove the compaction state information.

This algorithm does not need to stop to write new records on the physical medium, but, it needs some space at the end of the log to write compaction records. If there are enough records to remove, writing compaction records will only be needed at the compaction beginning.

VI Conclusion

$K_I{}^T L^O{}_G$ has three level of genericity. The first one is that of block abstract type. By choosing a storage organization and the semantics of block operations, a particular log management policy is selected. The second level, is that of the classes. Within a block abstract type and for each logging function, specific protocols can be used. The last level is that of the log configuration. The properties of a logical log are defined by the graph object nature and the way they are composed.

$K_I{}^T L^O{}_G$ is a good tool for policy experimentation. It is adaptable to application needs and system resources (characteristics and amount). The cost an application pays for its log is related to the functions it uses. However, describing an appropriate graph is a difficult task. Adding new classes requires recompilation the different service parts.

A prototype is currently under implementation. It is written in C and C++ over Unix (Sun OS).

The prototype is composed of a server, a client library, and a set of control programs. It implements only one block abstract type and one class of each kind of block (for medium block, only the disk class is written). The replication block is simple; it does not provide any consistency protocol. Compaction is not yet implemented.

Future work will begin by exercising $K_I{}^T L^O{}_G$ with real applications in order to do performance measurements and to validate the model and the log management policy.

Then, we plan to study new classes (specifically, the implementation of different replication protocols) and new block abstract types.

References

[Baer et al. 81] J. L. Baer, G. Gardarin, C. Girault, G. Roucairol: "The Two-Step Commitment Protocol: Modeling Specification and Proof Methodology". Proc. IEEE, 5^{th} International Conference on Software Engineering, San Diego, USA. 1981.

[Bernstein and Goodman 80] Philip A. Bernstein, Nathan Goodman: "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems". Proceedings of the Sixth Internationnal Conference on Very Large Data Bases. Pages 285–300. October 1980.

[Birman et al. 89] Kenneth P. Birman, Robert Cooper, Thomas A. Joseph, Kenneth P. Kane, Frank Schmuck: "ISIS - A Distributed Programming Environment, User's Guide and Reference Manual". pages 191–215. 19 June 1989.

[Black et al. 87] A. Black, N. C. Hutchinson, E. Jul, H. M. Levy, L. Carter: "Distribution and abstract types in Emerald". IEEE Transactions on Software Engineering, Vol. 13, N°1, pages 65–76. January 1987.

[Daniels et al. 87] Dean S. Daniels, Alfred Z. Spector, Dean S. Thompson: "Distributed Logging for Transaction Processing". Proceedings of ACM Special Interest Group on Management of Data. San Francisco. May 27-29, 1987. SIGMOD Record, Vol. 16, N°3, pages 82–96. December 1987.

[Daniels 88] Dean Spencer Daniels: "Distributed Logging for Transaction Process-

ing". PhD Thesis, CMU-CS-89-114. December 1988.

[**Druschel et al. 91**] Peter Druschel, Larry L. Peterson, Norman C. Hutchinson: "Lipto: A Dynamically Configurable Object-Oriented Kernel". Newsletter of the IEEE Computer Society Technical Committee on Operating Systems and Application Environments; Vol. 5, N°1, pages 11–16. 1991.

[**Finlayson and Cheriton 87**] Ross S. Finlayson, David R. Cheriton: "Log Files: An Extended File Service Write-Once Storage". ACM 11^{th} Symposium on Operating System Principles, Austin (TX); Pages 139–148. November 1987.

[**Gray et al. 81**] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzoli, Irving Traiger: "The Recovery Manager of the System R Database Manager". Communications of the ACM, Computing Surveys, Vol 13, N°2, pages 223–243. June 1981.

[**Haskin et al. 87**] Roger Haskin, Yoni Malachi, Wayne Sawdon, Gregory Chan: "Recovery Management in QuickSilver". Proceedings of the Eleventh ACM symposium on Operating Systems Principles; Austin, Texas; pages 75–86. 8-11 November 1987.

[**Heidemann and Popek 91**] John S. Heidemann, Gerald J. Popek: "A Layyered Approach to File System Development". Technical Report, University of California, Los Angeles, CSD-910007. March 1991.

[**Hutchinson and Peterson 88**] Norman C. Hutchinson, Larry L. Peterson: "Design of the x-Kernel". Proceedings of the SIGCOMM'88 Symposium. Stanford, CA (USA), pages 65–71. August 1988.

[**Leblanc and Mellor-Crummey 87**] Thomas J. Leblanc and John M. Mellor-Crummey: "Debugging Parallel Programs with Instant Replay". IEEE Transactions on Computers, Vol. C-36, N°4, pages 471–482. April 1987.

[**Ritchie 84**] Dennis M. Ritchie: "A stream input-output system". AT&T Bell Laboratories Technical Journal, Vol. 63, N°8, pages 1897–1990. October 1984.

[**Rosenblum and Ousterhout 91**] Mendel Rosenblum and John K. Ousterhout: "Design and Implementation of a Log-structured File System". Thirteenth ACM Symposium on Operating Systems Principles; Pacific Grove, CA, USA. ACM Operating Systems Review, Vol. 25, N° 5, pages 1–15. October 13-16, 1991.