BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M1

" Logging and Recovery in ObServer2"

by

Paul Alan Reilly

Logging and Recovery in ObServer2

Paul Alan Reilly
Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the Degree of
Master of Science in the Department of Computer Science at Brown University

February 1993

This research project by Paul Alan Reilly is accepted in its present form by the Department of Computer Science at Brown University in partial fulfillment of the requirements for the Degree of Master of Science.

Stanley B. Zdonik

Professor Stanley B. Zdonik
Advisor

2/17/93

Date

Logging and Recovery in ObServer2

Master's Project

Paul Alan Reilly

February 11, 1993

1. Introduction

This project is an implementation of the logging and recovery subsystem of ObServer2, a distributed persistent object store. This paper is concerned only with the logging and recovery system used in ObServer2 and presumes some knowledge of ObServer2. However, those with some understanding of database principles, should be able to follow most of it.. For a more detailed discussion of ObServer2 refer to David Langworthy's Ph.D. Thesis Proposal [2]. This project is an implementation of the logging and recovery system outlined in that proposal.

2. Overview

In order to understand various aspects and features of this project it is helpful to have an understanding of how the logging and recovery subsystem fits into the whole system. Figure 1 shows some of the relevant pieces of the ObServer2. Only those pieces on the server side of the system are shown. The logging and recovery system resides solely on the server side of ObServer2.

The major components include the Logging System which will be discussed in some

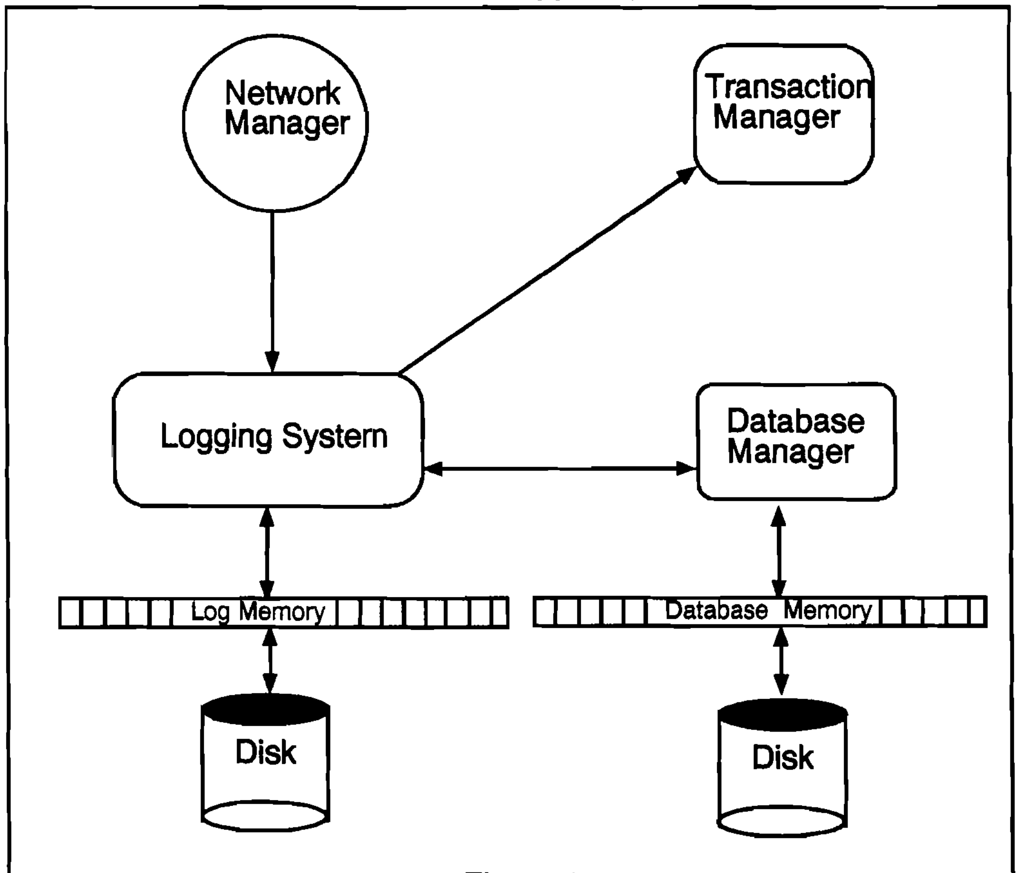

Figure 1

detail throughout this paper, the Network manager, the Database Manager, and the Transaction Manager. Each of these interact with the Logging System and must be understood as well.

The Network Manager is responsible for handling all communications between the client and server components of ObServer2. The Network Manager calls the appropriate Logging System routine whenever a logging function message is received from a client.

The Transaction Manager is responsible for managing the state of all transactions on this server. It must maintain which transactions are currently in progress, what state they are in, what objects are involved in those transactions, etc.. The Transaction Manager uses the Logging System to achieve these goals. The Logging System notifies the Transaction Manager whenever it receives operations which affect the state of the transaction (ie PREPARE and COMMIT). The Logging System also stores records of all operations on objects and the Transaction Manager uses these to determine whether or not a transaction can commit. The two systems work together during Checkpointing and Recovery to make sure that the state of Prepared and Committed transactions is preserved.

The Database Manager is responsible for storing the objects on disk. It works with the other components to supply these objects to the client and server when operations are being performed. It is the responsibility of the Database Manager to maintain stable copies of committed versions of objects. However, not every committed object state will be reflected in the database. The version of the objects stored in the database are considered base versions, to which the updates of committed transactions stored in the log must be applied to bring the object up to date.

Each of these components plays an important role in the ObServer2 system. However, this paper only deals with the Logging System implementation and its interactions with the other components, not the full functionality of these other components or their implementations.

3. Logging and Recovery

The logging and recovery system uses an original no-undo, no-redo write-ahead logging algorithm called No-Redo Write Ahead Logging (NR-WAL). Intention lists are used to eliminate redo during recovery. This allows recovery to be done in, at most, a single pass of the log from the last checkpoint onward. Also, the nature of the log allows updates to be applied asynchronously so that preparing and committing transactions are much quicker.

The following subsections explain various aspects of the Logging system. First the physical structure of the log is explained. Next, the major data structures used are outlined. Then the algorithms used for the major functions of the system are explained. Finally, the actual implementation is discussed including a list of all of the external and major internal interfaces of the system.

3,1. Log

The log is a circular log based on a memory mapped raw disk partition. This gives the log various features which include control over the actual layout of records on disk. Therefore, the cost of accessing sequential records is a fixed amount, not unknown as would be the case when using a file system. Also, the Logging System can synchronously write portions of the log to disk with assurance. Therefore, if the write succeeds, the data made it to disk.

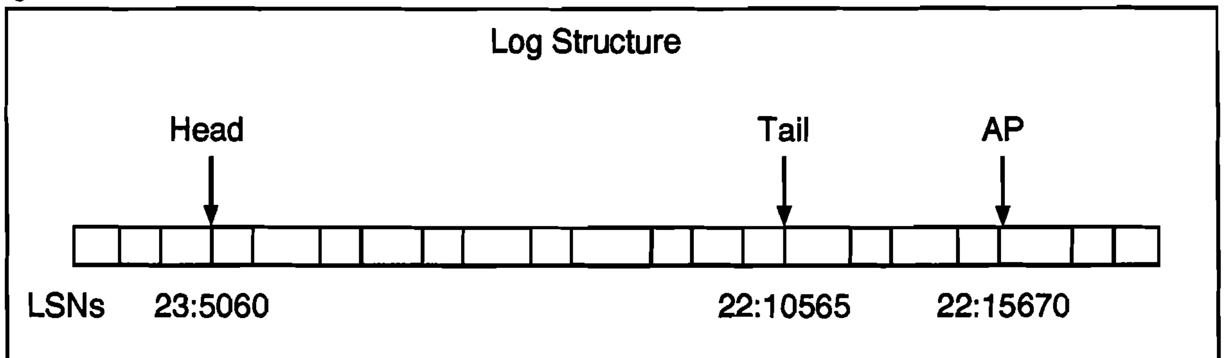

Figure 2

Figure 2 shows an example of what the log might look like. It is fixed in size. However, it can wrap-around to restart at the beginning as long as the data stored there is no longer needed. This allows the log to make better use of the space that it has while maintaining integrity of the relationships of LSNs (monotonically increasing).

3.2. Data Structures

The following are the important data structures used within the Logging System.

3.2.1. Log Sequence Numbers

A Log Sequence Number (LSN) is an index into the log. It is a montonically increasing value which will uniquely identify log entries. This allows the LSNs to be used for multiple purposes. The first purpose is to identify log records and allow them to be located. The second use is as a version number for objects store in the database. The LSN stored with the object is that of the log record for the most recent update operation. This allows LSNs to be used to see if an object is up to date.

An LSN consists of two parts. The first part, located in the high order bits, is a wrap count. This indicates how many times the log has wrapped to start back at the beginning. The second portion is a direct offset from the beginning of the memory of the log. This allows anything pointed to by an LSN to be reached without following indirect pointers.

3.2.2. Log Records

Every thing stored in the log is considered to be a log record. Each of these begins with a Log Record Header which contains the type of record, the lsn of the record itself, and the size of the record. The LSN of the record itself is used during the forward scan in recovery to detect whether the record is of the current wrap of the log. It is also used as a check for corrupt log entries. The size of the record allows records to be skipped while scanning the log.

Following the Log Record Header is the record itself. There are five types of log records - Intention, Checkpoint, Prepare, Commit, and NULL. Each of these has a different purpose and structure.

An Intention Record is used to record the operations, both observers and modifiers, that a transaction performs on various objects. This record consists of a Transaction Identifier, an LSN of the previous Intention record (used for linking together the Intention Records involved in a transaction), and one or more Operation Records which describe the operation being performed.

A Checkpoint record is used to store the state of the logging system. This allows the logging system a place to being upon recovery. After reading the checkpoint, the state is up to date with respect to transactions that prepared or committed before the checkpoint. The checkpoint record contains the transaction table, the version table, and their respective sizes.

A Prepare record is used to indicate that a transaction has successfully prepared (in the 2PC sense). The Prepare record contains a transaction identifier (TID), the LSN of the last Intention Record of the transaction and a timestamp (which is used to order transactions involved in blind updates).

A Commit record is used to indicate that a transaction has successfully committed. The Commit record contains a transaction identifier (TID), and the LSN of the Prepare record.

A NULL record is used as a filler in certain situations. The first case is when a record is to big to fit in the current wrap of the log. In this case a NULL record will be written out to take up the rest of the space of the wrap, and the record is written in the next wrap of the log. The next case occurs if a record needs to fit within a page, then a NULL record can be written out to take up the rest of the page (this has not been implemented yet).

3.2.3 Log Header

The Log Header is used to describe the state of the log. It is stored at the beginning of the log. Thus, it is a stabley stored piece of data. Whenever it is updated with data that is critical, it is synchronously written back to disk before proceeding. The Log Header contains a log identifier used to indicate that this is actually an ObServer2 log, a version of the format of the log, a flag indicating whether of not a checkpoint has ever been done, and the LSN of the last checkpoint.

3.2.4 Log Structure

The Log structure is a run-time structure which maintains volatile information about the log. The current Head, Tail, and AP are kept here. Locks controlling access to these are also stored here. Pointers to all of the routines registered with log_init() are stored in this structure. Also, the pointer to the actual log memory is stored in this structure.

3.3. Algorithms

This section outlines the algorithms used for various phases of the system. The four major sections considered are Logging Intentions, Checkpointing, Applying Updates and Recovery. Each of these will be described in turn.

3.3.1. Intention Logging

Intention Logging is fairly straight forward. When a client accesses objects, either observing or modifying them, it builds up an intentions list of these operations. These intentions lists contain transaction identifiers (Tid) which are used to identify the transaction. This allows the client to send the intentions list to the server whenever it wants. However, before a transaction can Prepare, all of the intentions lists from the client for that transaction must have arrived at the server.

When the server receives an intentions list, the intention list must be added to the end of the log. To do this a record header indicating the type of record (intention) and the size of the record is prepended, and the whole thing is written to the head of the log. After this occurs the log header is updated to indicate the new head of the log.

3.3.2. Checkpointing

The next major topic to be discussed is that of Checkpointing. During checkpointing, enough information is written to the log to ensure that the current state of prepare and committed transactions can be recovered. For NR-WAL, this information is fairly small. It consists of the Version Table, the Transaction Table, and the current head of the log. With this information the Recovery algorithm can reproduce the state of prepared and committed transactions.

Checkpointing consists of gathering this information and writing it to the log. To get the Version and Transaction Tables, the log manager calls the routines supplied to it upon initialization. This information is then added to the log with a log record header indicating the type of record (Checkpoint) and the size of the record. This information is then appended to the log.

However, this alone is not enough to guarantee that the state can be recover. The Recovery algorithm needs to know where the Checkpoint record is stored. Thus, after the Checkpoint record is written out (synchronously), the Log Header (a record at a known location) is updated with the LSN of the Checkpoint record. This guarantees that we can recover the state represented by the Checkpoint record.

3.3.3. Applying Updates

Applying Updates is the next major topic to be considered. In some ways this is simple for the logging system. All that the logging system worries about is, whether or not the intention record is of a transaction that has committed. Also, a check is made to see if there is an entry in the Version Table for this object. This would indicate that the stable state of the object may not be up to date with the committed state of the object. If so, then it can check to see if the object needs to have this update applied (by comparing the LSN of the object with the LSN in the Version Table).

Whenever an operation is found that needs to be applied, a routine which is registered for that operation code is called with the object, the operation code, and the arguments to the operation. It is up to this routine to apply the operation to the object. After this occurs the object can then be written back to disk with the updated LSN.

In order to implement this functionality, a thread, called the Applier thread, runs to apply these updates. It will continually check the log at the Application Pointer to check (by the above process) if there is an operation which needs to be applied. If the transaction has not committed, but is still active (ie not aborted), then the Applier thread waits. When the log receives a commit operation for a transaction, it signals the Applier thread after notifying the Transaction Manager. This will wake up the Applier, which can then check to see if the operation has committed. If the operation has aborted, then the Application Pointer is updated to the next operation. Otherwise the process continues as specified above.

An alternative implementation would be to have the Applier thread go through the log applying all updates that it can for committed transactions, without worrying whether or not they are the last one (ie pointed to by the Application Pointer) or not. Although this would apply more updates more quickly, it does nothing to free up the critical resource of log space because space can only be reclaimed at the Tail of the log. Thus, this approach does not seem worthwhile.

3.3.4. Recovery

The last major topic to be discussed is Recovery. Recovery is the process of restoring the state of the system as close as possible to the state that it was in before the crash, system failure, or shutdown. It is a requirement of the recovery process to be able to recover the state of any prepared or committed transactions. However, transactions which are not in these states may be lost.

When the logging system starts up (when log_init is called), the system reads the log header to find the last checkpoint. This checkpoint is then used to restore the state of the transaction and version table data which are passed to the Transaction Manager and Version Table respectively. Then a forward scan is made of the log to recover any thing that happened after the checkpoint was made. As prepare and commit records are encountered the Transaction Manager is notified with the information. After this scan, the system is ready to continue without any further work.

There are some interesting problems involved with this scan. The first is how do we know when to stop? One approach is that the scan continues until a record is found that is not valid (ie from the previous wrap, or has an invalid type, bad data, etc.). This approach has the major flaw that it will not discover any prepare or commit records which occur after this invalid record. Well, the question then arises can prepare or commit records occur after an invalid record. The answer is a modified yes. If only those records which are involved in transaction which is preparing are ensured to be synchronously written out to disk before a prepare occurs, then there could be records from other transactions intermixed with these records which will not be guaranteed to be written out. Thus, there could be an invalid record in between valid records and before a prepare record. If we used the above scheme for stopping, then these records would never be detected.

There are two solutions to this scanning problem. The first is to use a different criterion for determining when to stop scanning the log. One criterion would be to stop when the tail of the log is reached. However, this approach is extremely expensive. Not only does the whole log have to be scanned during a recovery, but whenever an invalid record is discovered, the log would have to be scanned byte-by-byte to ensure that the beginning of the next valid log record is found (think about a record which crosses pages to understand why this is so. If records can't cross pages, then we would just need to begin at the next page whenever an invalid record was found). A different approach would be to ensure that everything from the checkpoint to the last commit or prepare record was written out to disk. This would allow us to just scan forward until an invalid record is reached.

Unfortunately, neither of these solutions is currently implemented in the logging system. However, neither one would take long to add. Also, there are other possibilities for ensuring the correct recovery of the log, but only some straight forward ones were outlined here. If these are not satisfactory, other methods (like a mixture of the two above) could certainly be devised.

4. Implementation

The implementation of the log is fairly interesting. It uses a memory mapped partition (or file) to implement the log. This allows access to log records to be as easy as following memory pointers. This is especially critical during transaction Prepare when all of the intention records for a transaction must be examined for conflict, during observations when an object may have to be brought up to date, and during application of committed updates to the log.

4.1. External Interface

The interface to the log is as follows. Each of these routines is thread-safe. In other words, multiple threads can call these routines simultaneously and they will not cause incorrect results. Locks from the threads package described in Section 6 are used to control access to shared data.

Routine Name : **log_init**

Description:

Initializes the log upon startup. Will also start recovery when starting up after the log has already been initialized. Checks the beginning of the log for layout and then calls log_recover to do recovery.

Format:

```
int log_init(Log * log, char * partition,
        int (* vt_get) (void * * vt, unsigned * size),
        int (* vt_put) (void * vt, unsigned size),
        int (* vt_update) (Oid * oid, int state, Lsn * lsn),
        int (* tt_get) (void * * tt, unsigned * size),
        int (* tt_put) (void * tt, unsigned size),
        int (* tm_prepare) (Tid * tid, Lsn * last_ir),
        int (* tm_commit) (Tid * tid, Lsn * prep_ir))
```

Arguments:

log - pointer to log structure which contains information pertinent to the log. Most of this information in this structure is set in this routine

partition - character string indicating the file to use as the log. The file will be examined and its size will determine the size of the log. This does not actually have to be a partition, but using a partition may increase the performance (slightly).

vt_get - routine to be called when the log is going to do a checkpoint and wants a current copy of the version table. This routine must supply a version table that is in contiguous memory. The format does not matter except that it will be passed to the vt_put routine when a recovery is done.

vt_put - routine to be called when recovery is done. It gets passed a pointer to the version table that was stored during the last checkpoint.

vt_update - routine to be called when an update has to be made to the version table.. It gets passed a pointer to the object identifier (Oid), a state, and a pointer to the new LSN (if applicable).

tt_get - routine to be called when the log is going to do a checkpoint and wants a current copy of the transaction table. This routine must supply a transaction table that is in contiguous memory. The format does not matter except that it will be passed to the **tt_put** routine when a recovery is done.

tt_put - routine to be called when recovery is done. It gets passed a pointer to the transaction table that was stored during the last checkpoint.

tm_prepare - routine to be called during recovery when a PREPARE record is found. It gets passed a pointer to the transaction identifier, and a pointer to the last intention record lsn.

tm_commit - routine to be called during recovery when a COMMIT record is found. It gets passed a pointer to the transaction identifier, and a pointer to the prepare lsn.

Routine Name : **log_intention**

Description:

This writes an intention record to the log using the lower level log_add_record.

Format:

```
int log_intention(Log * log, Lsn * lsn,
                  Tid *tid, Lsn * prev_ir,
                  OperationRecord * op)
```

Arguments:

log - structure identifying log in which this record is to be written

lsn - returned lsn of where this record is placed.

tid - transaction identifier for the transaction of which this intention record is part

prev_ir - Lsn of previous intention record op - operation records

Routine Name : **log_checkpoint**

Description:

Places enough information into the log so that we can recover to the current state without reading the log. Need to write out the Version Table, Transaction Table, and Head of the log. This routine should probably be called periodically from a thread which sleeps. However, we may also want to be able to call it at specific points.

Format:

```
int log_checkpoint(Log * log)
```

Arguments:

log - pointer to the structure containing the info about the log that is being checkpointed.

Routine Name : **log_prepare**

Description:

Calls the registered prepare routine. Then, if the routine returns a success value, it will write out the prepare recrod and change the status in the various tables.

Format:

```
int log_prepare(Log * log, Lsn * lsn, Tid * tid, Lsn * last_ir)
```

Arguments:

log - pointer to the structure containing the info about the log in which to write the record.

lsn - place to write the LSN of the prepare record.

tid - pointer to transaction identifier of the transaction that is preparing.

last_ir - pointer to the LSN of the last intention record of this transaction.

Routine Name : **log_commit**

Description:

Writes out a commit record to the log. This will also force a log_sync for the log up to the end of the commit record of the transaction.

Format:

int log_commit(Log *log, Lsn * lsn, Tid * tid, Lsn * prepare_lsn)

Arguments:

log - Pointer to the log structure initialized by log_init().

lsn - pointer to Lsn structure which will indicate where this commit record is placed when it is written to the log. Filled out by this routine.

tid - Transaction identifier of the transaction which is committing.

prepare_lsn - Lsn of the prepare record of this transaction.

Routine Name : log_sync

Description:

Makes sure that a portion of the log is on stable storage.

Format:

int log_sync(Log *log, Lsn * beg_lsn, Lsn * end_lsn)

Arguments:

log - Pointer to the log structure which contains all information pertinent to the log.

beg_lsn - Pointer to the Lsn of the start of the section of the log to be stabilized.

end_lsn - Pointer to the Lsn of the end of the section of the log to be stabilized.

4.2. Internal Routines

The following set of routines are those that are used internally that are important. Each of these routines has a large impact on the system, and the understanding of them is vital to the understanding of the implementation of the logging and recovery system.

Routine Name : log_applier

Description:

This routine is started from log_init() after recovery is complete. It goes through the log trying to apply committed changes. It does this by looking at the Application Pointer and checking the records pointed to by it. If the record has been committed, a check is made to see if the change has already been applied. If not, then the registered routine is called with the OperationRecord contained in the Intention Record. After this has been done, the Appliction Pointer can be moved forward.

If the transaction represented in the record pointed to by the AP has not been committed, then a check to make sure that the transaction has not been aborted. If not, then the applier sleeps waiting for a transaction to commit. When a transaction does commit (or abort) , the transaction manager signals the applier (through the use of a condition variable) that a transaction has committed (or aborted), the applier starts up again. If the transaction aborted, then the AP can be moved forward as well.

Format:

void log_applier(void * temp)

Arguments:

temp - pointer to the log structure. Passed as a (void *) to satisfy C.

Routine Name : **log_add_record**

Description:

Low level routine to add to the circular log. It makes sure that the whole record will fit, including a log record header which is prepended by this routine. If the whole record will not fit in this wrap of the log, then a null record is written out, the log is wrapped, and the space that was remaining in the wrap is lost. If this record won't fit in the log at all, a negative value indicating the difference between available space and log record size. This routine takes a variable length arg list so that all the arguments do not need to be copied into one buffer for this routine to be called.

Format:

```
static int log_add_record(Log * log, LogRecordType type,
                Lsn * lsn, unsigned long total_size,
                unsigned long num_args,...)
```

Arguments:

log - Pointer to the log structure initialized by log_init().

type - type of record to be stored - *INTENTION, COMMIT,* etc.

lsn - lsn where the log record is placed, only valid if status is zero

total_size - size of all records to be placed in the log.

num_args - number of items to be written to the log And then once for each item:

size - size of the next item

buffer - pointer to the item in memory

NOTE: THIS ROUTINE SHOULD BE MODIFIED SO THAT IT TAKES AN ADDITIONAL PARAMETER WHICH INDICATES WHETHER OR NOT, THE RECORD CAN CROSS A PAGE BOUNDARY. IF NOT, AND IF THE RECORD DOES NOT FIT, WRITE A LOG_NULL *RECORD TO FILL THE REMAINDER OF THE PAGE. THIS IS ONLY IN THE CASE THAT THE RECORD WOULD FIT IN A PAGE TO BEGIN WITH. IF NOT, WE NEED TO DO SOMETHING DIFFERENT.*

Routine Name : **log_recover**

Description:

Routine which restores the state of the log as close as possible to the state before the crash/exit. This routine uses the last checkpoint and then scans forward to find all **PREPARE** and **COMMIT** records. It uses these to update the transaction and version tables. It can do nothing with any other record types due to the fact that we can't guarantee enough about them to be able to recover them. Thus, if a transaction was in process, but had not yet prepared before the crash, the transaction will be considered to have aborted after recovery.

This routine could be extended to continue reading forward even an invalid record is found. All that would be needed is to continue to scan through the log looking for the appropriate structure. However, this could be a lot more expensive due to the fact that we would have to look at every byte from the invalid record up to the tail of the log.

Format:

static int log_recover(Log * log, Lsn * cp_lsn)

Arguments:

log - Pointer to the log structure which contains all information pertinent to the log.

cp_lsn - pointer to the LSN of the last known checkpoint. This can be the NULL LSN if no checkpoints have been done.

5. Multiple Session TCP

5.1. Overview

One of the other packages that I wrote included a simple package to allow multiple simultaneous channels (or sessions) over a single TCP/IP connection. Actually it would work over any transport with a socket interface given relatively minor changes, but it only makes sense for connection-oriented protocols.

The reason for developing this package is that we need the ability to support multiple logical connections from client to server. Also, the number of these connections needs to be able to change during run-time. Although we could have used straight TCP/IP connections to achieve this, we felt that the cost for establishing separate connections was too high both in latency for establishing the initial connection and in cost of keeping each connection open.

The basic model of this package mimicks TCP/IP when initial connections are made. After that, everything is done with uni-directional session. The fact that sessions are uni-directional allows them to be created much more cheaply than if they were bi-directional.

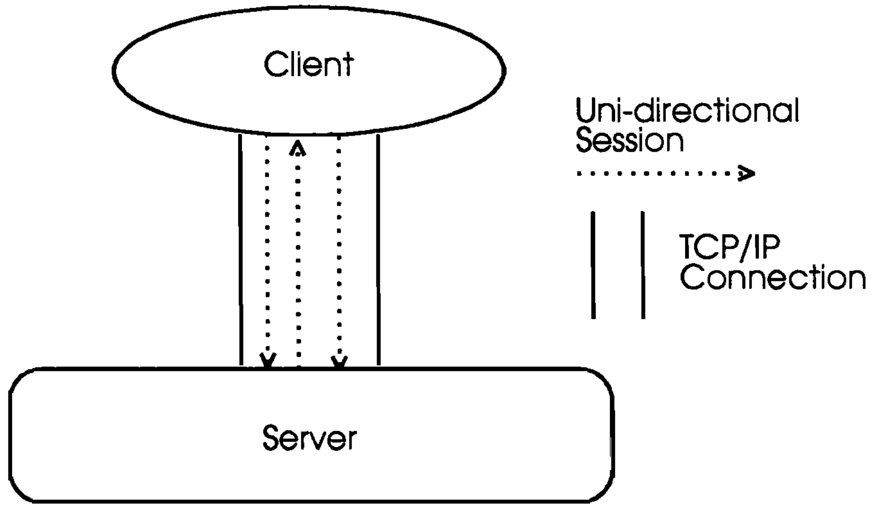

Figure 3

Figure 3 shows a sample configuration with one connection between the client and server. However, these connection has three logical one-way sessions within it. Two that go from the client to the server and one from the server to the client.

5.2. Interface

Routine Name : **create_server_end**

Description:

This routine is used on the server side to set up an initial endpoint for receing connection requests from clients. This routine is the one that actually sets up the TCP socket that is lused to accept connection requests. An inital session number 0 is created automatically which the client can write to and the server can read from.

Format:

int create_server_conn(Connection * conn, int sock)

Arguments:

conn - pointer to the log structure. Passed as a (void *) to satisfy C.

Routine Name : **create_client_conn**

Description:

This routine is used on the client side to set up an initial connection to the server.

Format:

int create_client_conn(Connection * conn, char * hostname, int port)

Arguments:

conn - pointer to the connection structure. filled in by this routine

hostname - hostname on which the server is executing

port - port of the server to be connected to

Routine Name : **create_session**

Description:

This routine is used on either endpoint and creates an endpoint that can be used to receive messages. Typically, one side will call this routine and then pass the session number to the other side to be used when responding to certain messages.

Format:

int create_session(Session * session, Connection * conn)

Arguments:

session - pointer to the session structure. Filled in by this routine

conn - pointer to the connection structure which was initialized by create_client_conn() or create_server_end().

Routine Name : **write_session**

Description:

This routine is used on the client end of a session to write to the end which created the session. This routine may block if the message can not be sent immediately.

Format:

int write_session(Connection * conn, Session * session, int type, int size, Message * msg)

Arguments:

conn - pointer to the connection structure which was initialized by create_client_conn() or create_server_end().

session - pointer to the session structure. Filled in by this routine

type - type of the message being passed. Used at the other end for whatever purpose it wants.

size - size of the message to be sent

msg - pointer to the message structure containing the message being sent.

Routine Name : **read_session**

Description:

This routine is used on the creating end of a session to read any messages sent to the session. This routine will block until there is a message received for this session.

Format:

```
int read_session(Connection * conn, Session * session, Message * msg)
```

Arguments:

conn - pointer to the connection structure which was initialized by create_client_conn() or create_server_end().

session - pointer to the session structure. Filled in by this routine

type - type of the message being passed. Used at the other end for whatever purpose it wants.

size - size of the message to be sent

msg - pointer to the message structure containing the message being sent.

6. Threads Abstraction Layer

6.1. Overview

This piece of the project was an attempy to isolate our use of threads from any particular threads package. To do this, we wanted to closely model something similar to the Posix threads package. The main requirements of a package needed are the ability to round-robin schedule the threads and some way to support non-blocking I/O (this is one of the main reasons that threads are being used in ObServer2). Other than that some way to create threads, and a way to create mutexes or locks is required. The condition facility could easily be implemented on top of this, but if provided reduces the amount of work needed to be done to replace the layer.

6.2. Threads

The threads abstraction is a simple model of threads which are round-robin scheduled. There is currently no way to control the size of the stack of the threads or the priority of the threads (they are all the same priority). This implementation uses Sun's Light Weight Process (LWP) pacakge to provide threads support. However, it would be very easy to use Brown Threads or Posix Threads to provide the same support.

Due to the fact that LWP threads are not preemptivelty scheduled, some additional work is needed. In order to support round-robin scheduling, a scheduler had to be used. This scheduler runs at a higher priority than all of the other threads and reschedules the threads every Quantum. This provides the effect of round-robin scheduling.

6.3. Interfaces

Routine Name : **threads_init**

Description:

This routine is used to do any initialization that is required for the threads package. There should also be a threads_rundown routine which would release all resources. However, this has not been implmented yet

Format:

int threads_init()

Arguments:

None

Routine Name : **thread_create**

Description:

This routine is used to create a new thread.

Format:

int thread_create(Thread * tid, void (* func) (void *), void * arg)

Arguments:

tid - pointer to the thread strucutre to be filled in by this routine.

func - pointer to the routine to be called when the thread starts up. The routine takes one argument which is specified by arg below.

arg - the argument passed to func upon startup of the thread.

Routine Name : **thread_yield**

Description:

This routine is used to yield the cpu to another thread.

Format:

int thread_yield()

Arguments:

None

Routine Name : **thread_join**

Description:

This routine is used to wait for a thread to exit.

Format:

int thread_join(Thread * tid)

Arguments:

tid - pointer to the thread structure of the thread to wait for.

Routine Name : **thread_sleep**

Description:

This routine is used to put a thread to sleep for a time.

Format:

int thread_sleep(struct timeval * timeout)

Arguments:

timeout - pointer to the timeval strucutre specifying the sleep duration.

Routine Name : **threads_exit**

Description:

This routine is used a program which uses threads to exit immediately. This occurs even if there are threads still running.

Format:

int threads_exit()

Arguments:

None

Routine Name : **lock_create**

Description:

This routine is used to initialize a lock (mutex).

Format:

int lock_create(Lock * lock)

Arguments

lock - pointer to the lock structure to be filled in by this routine.

Routine Name : **lock_acquire**

Description:

This routine is used to acquire a lock (mutex). It will block until it can get the lock.

Format:

int lock_acquire(Lock * lock)

Arguments

lock - pointer to the lock structure.

Routine Name : **lock_release**

Description:

This routine is used to release a lock (mutex).

Format:

int lock_release(Lock * lock)

Arguments

lock - pointer to the lock structure.

Routine Name : **lock_free**

Description:

This routine is used to free the resources associated with a lock (mutex). In other words, destroy the lock.

Format:

int lock_free(Lock * lock)

Arguments

lock - pointer to the lock structure to be freed.

Routine Name : **condition_create**

Description:

This routine is used to create a condition variable. A condition variable is used in conjunction with a lock. The lock must be acquired before any other condition functions can be called.

Format:

int condition_create(Condition * cond, Lock * lock)

Arguments

cond - pointer to the condition structure to be filled in by this routine.

lock - pointer to the lock structure to be used with this condition variable.

Routine Name : **condition_wait**

Description:

This routine is used to wait for a condition. The lock associated with the condition must be acquired before this function can be called. This routine will block. If it does, it releases the lock, until the condiation is satisfied. However, the lock is reacquired before returning.

Format:

int condition_wait(Condition * cond)

Arguments

cond - pointer to the condition structure to be waited for.

Routine Name : **condition_notify**

Description:

This routine is used to signal a condition. The lock associated with the condition must be acquired before this function can be called.

Format:

int condition_notify(Condition * cond)

Arguments

cond - pointer to the condition structure to be notified

Routine Name : **condition_free**

Description:

This routine is used to release the resources associated with a condition. The lock associated with the condition must be acquired before this function can be called.

Format:

int condition_notify(Condition * cond)

Arguments

cond - pointer to the condition structure to be freed

Makefile

7. Modules

The following is a list of modules and what they are used for:

Makefile - used to build the system

log.c - contains all of the logging routines

log.h - contains all definitions for the logging routines

mtcp.c - contains all of the routines for the multiple session tcp/ip, and the threads package.

mtcp.h - conatins all of the definitions for the mtcp and threads stuff.

client.c - client side test program for mtcp and threads.

server.c - server side test program for mtcp and threads.

log_test.c - test program for some simple log functions.

8. References

1. Franklin, Michael J., Zwilling, Michael J., Tan, C. K., Carey, Michael J., DeWitt, David, Crash Recovery in Client-Server Exodus. In *ACM SIGMOD*, 1992.

2. Langworthy, David E., ObServer2: Extensible High Performance Support for Persistence, *Ph.D. Thesis Proposal (Draft)*.

3. Mohan, C., Haderle, Don, Lindsay, Bruce, Pirahesh, Hamid, and Schwarz, Peter, ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, In *ACM TODS*, March 1992.