Unique Atomic Chunks - A Pattern for Security and Reliability

Abstract

This paper presents a security and reliability pattern that describes an alternative to ensure reliable writing over a shared resource by multiple processes. Multiple processes writing over a single file are often vulnerable to crashes and because of unreliable non-atomic commits traces of crash are often visible. Writing in a directory (creating and updating a new file) provides better services for locking. Therefore creation of new files ensures more reliability.

Unique Atomic Chunks

Reliability, Security

Pattern Name
Unique Atomic Chunks

Intent
When multiple processes try to write some data concurrently to the same resource, there is need for some locking mechanism. But the locking mechanism is not always atomic and a crash may leave a trace in the resource. It can be useful to create separate entry for each information write request - if the separate entries are within a manageable limit.

Motivation
This can particularly be useful in creating a mailing directory. In UNIX, traditional Mail Transfer Agents (MTA) like 'sendmail' follow a particular mailing directory format called the 'mailbox' format. This format, along with a latter modification named 'mbox', use a single file for all mail entry. So, when multiple sources are trying to put their mail into the file, there is some need for locking the resource. However, a machine may crash while it is delivering a message. For the traditional formats, this means that the message will be silently truncated. Even worse: for '*mbox'* format, if the message is truncated in the middle of a line, it will be silently joined to the next message. The MTA will try again later to deliver the message, but it is unacceptable that a corrupted message should show up at all.

Ideally, every message should be guaranteed complete upon delivery. A machine may have two programs simultaneously delivering mail to the same user. The traditional formats require the programs to update a single central file. If the programs do not use some locking mechanism, the central file will be corrupted. There are several locking mechanisms, none of which work portably and reliably.

Many sites use Sun's Network File System (NFS). NFS exacerbates all of the above problems. The locking mechanism known as 'dotlocking' is popular in a world filled with NFS file systems (a problem in its own right), because ordinary file locking tends to create all sorts of problems with hung state and lock daemons and stuff. Some NFS implementations even don't provide any reliable locking mechanism. With *traditional* formats, if two machines deliver mail to the same user, or if a user reads mail anywhere except the delivery machine, the user's mail is at risk.

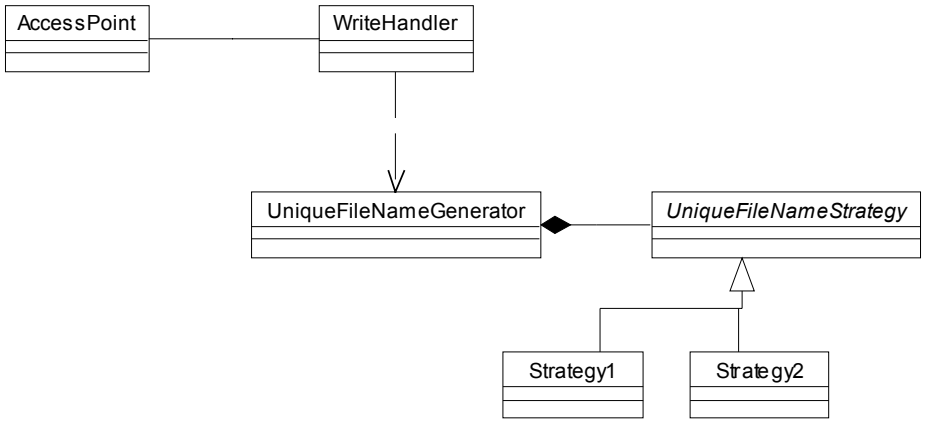
How do we ensure that in this situation, multiple concurrent requests are handled correctly and even if there is a crash no trace is left of the failure?

A better solution is to create separate and unique file entries each time a write request occurs. Again, these entries can be kept at different directories and moved from one directory to another during different phases of mail delivery process. A Mail User Agent (MUA) can read and delete messages while new mail is being delivered. Because, each message is stored in a new, unique file, it does not affect other operations. Different delivery processes never touch the same file – so there is no reason that the remnants of crash will be existent.

Applicability
Use this pattern when

- Multiple sources are trying to access a single file for write. (Add or Delete)
- The locking mechanism does not work effectively for handling concurrent calls.

Structure
Here is an object-oriented presentation of the structure.



Participants

- **AccessPoint**
  - A policy enforcement point where the requests for write come in.
- **WriteHandler**
  - Handles the write task
- **UniqueFileNameGenerator**
  - A Singleton (GoF) for generation of unique file name.
- **UniqueFileNameStrategy**
  - Strategy (GoF) pattern for unique filename creation algorithms

Collaborations
Different sources put their write request for write access. The AccessPoint class handles the request by delegating them to WriteHandler. The WriteHandler class uses a Singleton UniqueFileNameGenerator to create unique file name for the request. The unique file name creation is the most crucial issue of this pattern. A system can have several strategies to employ – most notably for different platforms and file systems. This is implemented as Strategy.

Consequences
The pattern has the following benefits and liabilities

1. *It provides reliable write task*. Because separate threads are employed in creating unique files, this provides reliable task completion. Unit of Work [PEAA] pattern can be employed in the WriteHandler class for ensuring reliability. Different phases of the task can leave the system in different states. Only after all the steps are committed can the task dubbed as 'done' (thus reaching the final state). So partial write and crashes are not an issue anymore. Besides, crashed files are recreated again upon retry, so there is no unwanted trace of previous system failure.

2. *Atomic content writing*. The key issue is the atomicity in writing messages. File writing or appending is unreliable because the underlying file system does not provide atomic locking mechanism. However, the creation of a file is an atomic process enforced by the file system.

Creating a new file is also in essence a write in the file system except for the fact the content is written in the directory hierarchy (the buffer cache). Therefore, when a new, unique file is created under a directory a lock on the directory is placed. Network File System (NFS) makes sure that this locking and file creation under a directory is an atomic process. This mechanism is tidier than placing a lock on a file and ensuring the commit job is atomic.

3. *Unique File Name.* The files are created by different processes. There should be some mechanism to ensure that multiple processes do not contend over a filename. This separation is very important for ensuring reliability. In Unix, a lot of strategies can be adopted to achieve this. They are described in the implementation section.

4. *Manageable File Count*. Since, all the write requests employ new file creation, the number of files can quickly become unmanageable. That is why, it is not useful if write requests occur more frequently than a manageable limit. A mail system is an ideal example of such a scenario. This also necessitates a size constraint and file count constraint and therefore garbage cleaning and removal policy. (See more in the Implementation section).

5. *Performance issue*. Even with garbage collections, as file count increases, searching for data can be a performance issue. This is because in Unix, normally finding a file in a directory becomes slow if that directory contains many files. Creating subdirectories under one directory and storing files under that subdirectory may provide a solution. This is described in detail in relation to qmail queue management.

Implementation

We describe here two implementation scenarios of this pattern. Also we cover the issues already mentioned and how they are handled in these implementation scenarios. Finally we discuss the issues involving Windows implementation.

1. *Unix implementation of 'maildir'*. The motivation section describes the problem of a mail directory acting as a shared resource in a publish-subscribe channel. qmail follows this pattern in the 'maildir' structure. A 'maildir' is a structured directory that holds e-mail messages. qmail's 'maildir is a simple data structure, nothing more than a single collection of e-mail messages. A directory in 'maildir' format has three subdirectories, all on the same filesystem: tmp, new, and cur.

Each file in new is a newly delivered mail message. The modification time of the file is the delivery date of the message. Files in cur are just like files in new. The big difference is that files in cur are no longer new mail - they have been seen by the user's mail-reading program. The tmp directory is used to ensure reliable delivery, as discussed here.

A program delivers a mail message in six steps.

First, it chdir()'s to the 'maildir' directory.

Second, it stat()'s the name tmp/time.pid.host, where time is the number of seconds since the beginning of 1970 GMT, pid is the program's process ID, and host is the host name.

Third, if stat() returned anything other than ENOENT, the program sleeps for two seconds, updates time, and tries the stat() again, a limited number of times.

Fourth, the program creates tmp/time.pid.host.

Fifth, the program NFS-writes the message to the file. NFS-writing means
(1) checking the number of bytes returned from each write() call
(2) calling fsync() and checking its return value
(3) calling close() and checking its return value

Sixth, the program link()'s the file to new/time.pid.host. At that instant the message has been successfully delivered. See sample code in qmail.

2. *Unix implementation of mail queue of qmail*. The queue of qmail is managed by three processes. The queue is a shared between qmail-queue (inserts messages into the queue), qmail-send (reads messages from the queue and takes appropriate sending actions) and qmail-clean (garbage collection).

The mail queue is divided into multiple directories and messages are moved from one directory to another. This transfer process from one directory to another resembles different states in a message queuing cycle. All the time the state information is retained.

qmail's queue consists of several directories. The directory structure is as follows.

Subdirectory	**Contents**
bounce	permanent delivery errors
info*	envelope sender addresses
intd	envelopes under construction by `qmail-queue`
local*	local envelope recipient addresses
lock	lock files
mess*	message files
pid	used by `qmail-queue` to acquire an i-node number
remote*	remote envelope recipient addresses
todo	complete envelopes

Directories marked with an "*" contain a series of split subdirectories named "0", "1", ..., up to (conf-split - 1), where conf-split is a compile-time configuration setting (usually a prime number) contained in the file conf-split in the build directory. The newly created file is put in directory number (inode mod conf-split).

To add a message to the queue, qmail-queue first creates a file in a separate directory, pid/, with a unique name. The filesystem assigns that file a unique inode number. Usually, the unique file names will come from the unique inode numbers. qmail-queue looks at that number, say 457. The system is now in state 1. Files named as inode numbers also solves the problem of multiple processes trying to somehow create one single file. In this case, the losing processes in the race would have to create filename with a different inode.

qmail-queue then divides the inode number with conf-split value if the value is present in the configuration files. Let us suppose, the default value of conf-split is existent. The modulus of the division 457 by 23 is 20. So, qmail-queue renames pid/457 as mess/20/457, moving to State 2.

It writes the message to mess/20/457. It then creates intd/20/457, moving to State 3, and writes the envelope information to intd/20/457.

Finally qmail-queue creates a new link, todo/457, for intd/20/457, moving to State 4. At that instant the message has been successfully queued, and qmail-queue leaves it for further handling by qmail-send.

When qmail-send notices todo/20/457, it knows that message 457 is in State 4. It creates info/20/457, possibly local/20/457, and possibly remote/20/457. When it is done, it removes intd/20/457. The message is still in State 4 at this point. Finally qmail-send removes todo/20/457, moving to State 5. At that instant the message has been successfully preprocessed.

Each address in local/20/457 and remote/20/457 is marked either NOT DONE or DONE.

> **DONE:** The message was successfully delivered, or the last delivery attempt met with permanent failure. Either way, qmail-send should not attempt further delivery to this address.
>
> **NOT DONE:** If there have been any delivery attempts, they have all met with temporary failure. Either way, qmail-send should try delivery in the future.

qmail-send may at its leisure try to deliver a message to a NOT DONE address. If the message is successfully delivered, qmail-send marks the address as DONE. If the delivery attempt meets with permanent failure, qmail-send first appends a note to bounce/457, creating bounce/457 if necessary; then it marks the address as DONE.

When all addresses in local/20/457 are DONE, qmail-send deletes local/20/457. Same thing happens for remote/20/457. When local/20/457 and remote/20/457 are gone, qmail-send eliminates the message. qmail-send deletes info/20/457, moving to State 2, and finally mess/20/457, moving to State 1.

Now we discuss how different issues are handled.

a) Reliable write task and no mail loss. In qmail queue, because all the messages are kept in the file system with unique inodes and named as inode numbers, the messages survive crash because they are not in memory but permanently residing in file system. If the whole file system crashes, then the data would be lost but there is no expectation of retrieval in that case anyway. Besides the queuing process goes through separate states and there are different error handling paths for different states.

For example, if the computer crashes while qmail-queue is trying to queue a message, or while qmail-send is eliminating a message, the message may be left in state 2 or 3. When qmail-send sees a message in state 2 or 3, it deletes intd/20/457 if that exists, then deletes mess/20/457. Any qmail-queue handling the message must be dead at that point.

Cleanups are not necessary if the computer crashes while qmail-send is delivering a message. At worst a message may be delivered twice. There is no way for a distributed mail system to

eliminate the possibility of duplication. What if an SMTP connection is broken just before the server acknowledges successful receipt of the message? The client must assume the worst and send the message again. Similarly, if the computer crashes just before qmail-send marks a message as DONE, the new qmail-send must assume the worst and send the message again. This redundancy is not harmful in any way, but it is very crucial in ensuring the reliability of the whole system.

b) Unique File Name Creation. In the second step of 'maildir' implementation, the unique file-names are created by appending time, pid and host. Various schemes of concatenation of any number of following strings have been employed by modern delivery identifiers to absolutely guarantee the uniqueness.

- #n, where n is (in hexadecimal) the output of the operating system's unix_sequencenumber() system call, which returns a number that increases by 1 every time it is called, starting from 0 after reboot.

- Xn, where n is (in hexadecimal) the output of the operating system's unix_bootnumber() system call, which reports the number of times that the system has been booted. Together with #, this guarantees uniqueness; unfortunately, most operating systems don't support unix_sequencenumber() and unix_bootnumber.

- Rn, where n is (in hexadecimal) the output of the operating system's unix_cryptorandomnumber() system call, or an equivalent source such as /dev/urandom. Unfortunately, some operating systems don't include cryptographic random number generators.

- In, where n is (in hexadecimal) the UNIX inode number of this file. Unfortunately, inode numbers aren't always available through NFS.

- Mn, where n is (in decimal) the microsecond counter from the same gettimeofday() used for the left part of the unique name.

- Pn, where n is (in decimal) the process ID.

Old-fashioned delivery identifiers use the following formats:

- n, where n is the process ID, and where this process has been forked to make one delivery. Unfortunately, some foolish operating systems repeat process IDs quickly, breaking the standard time+pid combination.

- n_m, where n is the process ID and m is the number of deliveries made by this process.

In qmail queue, the file names are named as inode numbers. This ensures the uniqueness of their names.

c) Managable File Count. In 'maildir', The delivery program is required to start a 24-hour timer before creating tmp/time.pid.hos*t,* and to abort the delivery if the timer expires. Upon error, timeout, or normal completion, the delivery program may attempt to unlink() tmp/time.pid.host. Thus, resource is kept to a manageable limit.

For qmail queue, files in the queue are time-stamped once they are inserted by qmail-queue. If some file is not sent after a certain limit (specified during build phase), qmail-clean deletes them anyway. This performs garbage collection in the worst case.

Garbage collection is important for another reason here. Because the number of inodes is limited, some process has to free up the inodes for future file creation. qmail-clean is especially useful for this reason.

d) Performance issue. The use of conf-split in creating files under different subdirectories (of queue) improves file search time if the number of files is very large.

e) Windows implementation. The parallelism in Windows system is achieved through the use of threads. Unlike processes, threads do not have private address space. In this case unique file name generation can be achieved by using Singleton (GoF). Most of the Windows based MTAs, however, keep the messages in one file and rely on the locking and synchronicity mechanism provided by the thread. In windows systems, this pattern is used in some other cases of resource management. They are outlined in the known uses section.

Sample Code
Here is the UNIX implementation code for qmail maildir implementation.

.....
.....

//Step 1 - Moving to maildir directory

```
if (chdir(dir) == -1) { if (error_temp(errno)) _exit(1);
_exit(2); }
```

//Getting pid and hostname

```
pid = getpid();
host[0] = 0;
gethostname(host,sizeof(host));
```

//Step 2-4 - Main loop for unique file creation

```
for (loop = 0;;++loop)
 {
  time = now();
  s = fntmptph;
  s += fmt_str(s,"tmp/");
  s += fmt_ulong(s,time); *s++ = '.';
  s += fmt_ulong(s,pid); *s++ = '.';
  s += fmt_strn(s,host,sizeof(host)); *s++ = 0;
  if (stat(fntmptph,&st) == -1) if (errno == error_noent)
break;
  /* really should never get to this point */
  if (loop == 2) _exit(1);
```

//Wait for an interval to try again in case of a stat() result other than ENOENT

```
  sleep(2);
 }
```

//Step 5 - Copies the file – NFS writing

```
str_copy(fnnewtph,fntmptph);
byte_copy(fnnewtph,3,"new");
……
……
substdio_fdbuf(&ss,read,0,buf,sizeof(buf));
substdio_fdbuf(&ssout,write,fd,outbuf,sizeof(outbuf));
……
……
if (substdio_flush(&ssout) == -1) goto fail;
if (fsync(fd) == -1) goto fail;
if (close(fd) == -1) goto fail;
```

// Step 6 - Links the file

```
if (link(fntmptph,fnnewtph) == -1) goto fail;
……
……
```

Known Uses

This pattern is particularly useful for mail message handling. This pattern was used in qmail with considerable success. qmail's queue and mailboxes (MailDir) use this pattern. Later, MailDir format was accepted, employed and supported by POSTFIX, EXIM, COURIER and all other important MTA's. It has also been used with Mac OS X Mail Application with RCI mail server. Postfix queues also follow this pattern.

Attachment systems of Eudora Pro uses this strategy. If there is a failure in downloading the attachment then upon next try the system creates a new file name.

Caching and storage of temporary Internet files also employ this strategy.

Related Patterns

Singleton and Strategy patterns have been used for its implementation. Unit of Work pattern can be used for ensuring reliability.

Reference

1. [GoF] Design Patterns: Elements of re-usable object-oriented software – Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
2. [PEAA] Patterns of Enterprise Application Architecture – Martin Fowler
3. qmail MAN pages