Aim

Build a publisher-subscriber data streaming server that is scalable, persistent, redundant, robust, and pays particular attention to optimizing throughput and latency at the expense of loosing the very latest messages still in transit.

Platform

For performance and code simplicity, we forgo platform independendence and stick to linux. We use the following OS features, some C++ STL containers, and no third party libraries:

futex() for contended locking / semaphores

mmap() for persistence

epoll() for event driven IO

filesend() for zero-copy kernel-side cache/DMA file-to-socket transfers

pthread_create() to spawn more worker threads if there is no waiters on epoll_wait()

pipe() for main-thread / worker thread communication.

Design Considerations

Scalability - Horizontal vs. Vertical

While we want to make the data center scalable by growing more nodes, it would be nice if each node was operating at high capacity. In particular, if one server could handle all the write operations for a single busy stream, it would reduce system complexity greatly, compared to totally distributed writes.

It can still have other servers that it replicates to which can help distribute to many subscribers. As each subscribing server can have subscribing servers of its own, within 4 hops at 8 splits at each hop, it has propogated to 4096 leaf slaves. These could each accept many client connections. For the master and intermediate slaves we kept the splits down to 8 in this example because they are likely splitting many other streams too and we don't want a master (of many streams) to get bottleknecked by his connection to the data center, so we split fewer ways and hop more times as the latency in the data center is very low anyways. But for those 4096 leaf nodes in this example, each could have thousands fo subscribers.

To make a single-master-per-stream model possible, we need high peformance nodes. To maximize throughput to the disk, we want to only schedule pages for synching, but not wait for the sync to happen - M_ASYNC. We want the disk scheduler to have maximum flexibility to minimize disk seeks. The downside is that in case of a hardware or kernel failure we cannot rule out corruption. Even though we take care to write the data and then increase the varmap.len, the disk scheduler may write those in the opposite order, and if power fails in between, we have some length at the end of the file which contains nulls (all 0), event though we think it should contain data.

We can accept this because, with push subscriptions, slave servers in the data center will have been almost if not completely up to date. An automatic switch to promote one to master is the fastest recovery.

Networking – upstream vs. downstream sockets

Each stream on each server is either disabled, master, or slave. If it is slave, it is slave to either the master or another slave. Wether it is master or slave, it can have many slaves of its own. Slave very simply means that the server has gone out to another server over a socket, just like an outside client would, and has subscribed to its stream on that socket. So it caught up and then went into push mode. As soon as a new message is published by the master, it will get pushed all the way down the subscriber tree, servers and clients. The fact that slaves can have slaves means that exponential distribbution is possible without overloading the master with too many direct subscriptions.

Upstream sockets are those from the server to other servers to slave their streams. Downstream sockets are those that the server got through an accept() on it's listening socket. Through them the server gets subscription requests, and other commands.

Here is all you need to know:

1. commands go upstream, including “subscribe me to something...” as well as, “publish this message...”
2. stream content flows downstream.

This means that if a command to publish a message hits a slave server, it cannot get it into the stream as the stream is coming down from the master. Not really. While it cannot get it into the stream, because the slave has an upstream connection to the master directly or through other slaves, it can simply forward publish commands for streams that it is slaving. Remember that commands go upstream. So the same socket that is bringing it the stream is used to forward a message up all the way until it hits the master. This means that if the load distributor gets the message to any stream slave, it will get forwarded up to the master within the data center, timestamped and sequenced, and then broadcast propogated back down to all slaves.

Persistence

varmap.h is a C implementation of a vector, similar to STL::vector<char>, that uses mmap() instead of malloc() and mremap() instead of realloc(). The vector struct itself, struct varmap, resides at the beginning of the mmap'ed buffer. This struct contains the true length of the buffer, beyond which is just unused space (garbage). Care is taken to always do append type stuff first and then increase the length if all goes well. This way, even with M_ASYNC, even if the server crashes, so long as the kernel does not crash or the hardware does not go down, everything up to the recorded length will be legitimate data, not garbage.

But because we want to maximize throughput, for kernel/hardware crashes we transfer the responsibility of recovery to slave servers and run with M_ASYNC. However, this could simply be changed to M_SYNC instead and the system would drop probably substantially in throughput but be totally recoverable. It seems to be a big loss and a small win though as system failure could mean total drive failure, and unless buying expensive redundant drives, it's better to rely on replication to slaves for recovery, with ultra low push latencies within the data center.

Robustness

Whatever the cause of failure, be it that the master blows up, or a slave needs to be rebuilt, or a TCP connection within the data center goes down and reconnects, forcing slaves to resubscribe, the solution is always starting some subscriptions somewhere.

Because the master, and intermediate slaves, guarantee stream sequence, restarting is as simple as specifying to upstream as of when in time to start from.

Sub 25 12345

In this case we subscribe to stream 25, as of 12345 being the unix timestamp but in microseconds (64 bit).

We could have specified where to pickup from by the index into the array. But that approach suffers if we do a maintenance left-trim of the stream to get rid of old stuff, invalidating indecies. Of course we can keep track of how much has been removed, but the simpler, more robust solution was to just use a usec precision gettimeofday() which on my system seems to take 2 to 3 usec to return. So it actually has great resolution, but also not so much that we will get collisions. In other words, we should not end up with the same time value for 2 messages in the same stream very often. With “thousands of messages per stream per second” and hundreds of thousands of time values that could be returned per second, collisions should be rare. Using time is faster to code and more robust, but has the disadvantage that if two messages where received into the same stream resulting in identical time values, and a failure were to occur on a subscriber after the first but before the second. Upon resubscribing, we have to decide if to include that last time or not to include it. If we include it in the history download, we may get the first record twice. If we don't we may not get the second record of that time at all. A workaround is for the subscriber to say how many of records at that given time he has, most always being 1. But we skip that for now.

Stream Disk Layout

For high throughput we consider that a stream is basically an append only application. Exception is of course possible maintenance trimming of older data (left-trim). Given this, we would not benefit from more generalized database systems and would only suffer performance and complexity.

So we basically use memory-mapped append only flat files as follows.

A stream is a sequenced set of messages (sequenced by the server in the order they got timestamped). They are back to back in bytes, each starting with a header that defines the total length. This is in the .content file for the stream in the running directory. There is a matching .indx file. That one is the same sequence, but one fixed size entry per message back to back which contains timestamp, and locates the message in .content with an offset. The timestamps will be ascending (baring setting the clock back on the server which will have minor side effects). The .indx file is used to find a point in time (binary search) and get the matching .content offset.

It gives us constant time inserts (always append of course with low disk seek frequency as writes are always made to the end). Also, the .indx used for binary searching is much smaller (no message bodies) and we can fit much more or all of it in RAM. In particular, because it is being used in conjunction with a binary search, the index entries on the high order power-of-two offsets (half file_len, then 1/4 and 3/4, then 1/8, 3/8, etc.) will be in the OS file cache as they are high freq. hits. If the index is very big, only the leaf nodes plus their parents would constitute (1/2 + 1/4) = 3/4 of the mass of the index. Meaning it would only cost 2 disk seeks in the worst case (instead of log n disk seeks) with only 1/4 of the index in RAM.

This model, in particular makes the handlying of the most important use case, the subscription, simple, high-throuput, and it does not block the stream push to already subscribed sockets as explained below.

High Throughput / Low Latency

For low latency we accept that we need a push architecture. Meaning that when a message arrives into the stream, all subscribers will be sent a copy to their TCP buffers immediately, unless their own socket is locked, most likely by a bulk send operation they requested, in which case it will be buffered in sequence.

But when a subscriber connects and wants some or all history before getting the new messages, this bulk download must not block the stream.

1. binary search sorted list (in corresponding indx file) in O(log n) time to find an entry by time. Note the offset and also the current file len.
2. unlock the stream for further appends by concurrent threads using the mmap'ed memory handled by varmap.h.
3. Send using sendfile() (cache or DMA straight from file to socket in kernel space) on the underlying file descriptor from the noted file offset to the known file len before the unlock. Don't touch the mmap'ed memory directly because it may get mremap'ed by the other threads as the file grows.
4. lock the stream again to see if there have been indeed more entries appended. First time checking this on a busy stream will turn up more entries likely. If so, go back to 2, this time sending the newly arrived entries. Get their file offset and len, then repeat from 2 until we are up to date.
5. with the stream lock held from step 4 now that we are caught up, add the socket to the stream subscribers, before unlocking the stream.

This ensures that the socket gets offline high throughput catchup and then joins the stream without missing a single message or blocking the stream.

Write buffering

Primarily we try to use the TCP buffer. If it fills up and a non-blocking write is unable to write data, we just drop the connection. This is necessary as we can't possibly buffer for non-responsive clients endlessly. So we might as well just increase the TCP buffer's size to something appropriate and use that as the cutoff. The receiver will have to not subscribe to more things than it can remove from the network, otherwise he will quickly fill up and get disconnected.

Since the writes will not block, everyone can just get the lock_socket() and there shouldn't be sleeping going on for any duration. But, there is a reason why we will still block. The initial history download can actually block on disk, and since it's a kernel level file to socket operation, we socket_lock()'ed for the duration, meaning that blocking on disk will block that socket even if the TCP buffer is empty. So we do need another buffer.

The send_queue is that buffer. It is unordered_map<int socket, vector<char> *v>. There will be at most one vector for any socket in the send_queue, only if the socket itself has been locked, in which case the holder of that lock will flush the send_queue buffer to the TCP buffer before he calls unlock_socket(). Therefore, there is no down time sitting in the queue waiting for a worker thread to ship. As soon as the socket is available, the send_queue buffer for that socket is sent by the releasing thread.

To implement this, these are the 2 different write methods, used depending on whehter it is a kernel

blocking operation or a regular send from a RAM buffer:

1. Blocking operation: sendfile()

```
ssize_t file_send(struct engine *engine,
                  const int socket,
                  const int fd,
                  const off_t offst,
                  const size_t len)
{
   lock_socket(engine, socket);
   set_socket_block(socket, 20*1000*1000); //block for 20 sec, Think about DOS!!
   const size_t rslt = file_send_core(socket, fd, offst, len);


   lock_socket_buf(engine, socket);
   socket_flush(engine, socket);
   set_socket_non_block(socket);
   unlock_socket(engine, socket);
   //now noblock_send() would get the socket lock itself
   unlock_socket_buf(engine, socket);
   return rslt;
}
```

Note the careful handing off of responsibility. This is not just a point of optimization. There is no worker thread going through the send_queue. The thread that had the socket locked is responsible for sending anything that has piled up. By emptying the queue, unlocking the socket, and then unlocking the socket_buf, it says that while I had the socket, the socket_buf did not get forgotten. If everyone can say that, then there is never anything left waiting in the socket_buf.

A point about locking and concurrence: To avoid any blocking, socket_flush() does this:

```
void socket_flush(struct engine *engine, const int socket)
{
   lock_send_queue(engine);
   while(1) {
      vector<char> *v = NULL;
      auto it = engine->send_queue.find(socket);
      if(it == engine->send_queue.end()) { break; } //until nothing left
      v = it->second;
      engine->send_queue.erase(it);
      unlock_send_queue(engine);
      unlock_socket_buf(engine, socket); //so other's can buffer
      if(v) {
         send_bytes(socket, &(*v)[0], v->size());
         delete v; v=NULL;
      }
      lock_socket_buf(engine, socket);
      lock_send_queue(engine);
   }
   unlock_send_queue(engine);
}
```

The send_queue and the socket_buf are made available for new threads needing to buffer send to the socket while the old buffer is being sent to the TCP buffer. Then the locks are reapplied and the send_queue is queried for a potentially new buffer, until there is none left. In other words, we go in with the locks applied, we come out with the locks applied, and in between we flush buffer guaranteeing empty on the way out. BUT, for the duration of a flush, we temporarily let go of locks and allow a new buffer be made, flushing the old one offline, and looping if another was made.

2. noblock_send()

This is the counterpart function used to write the push subscription data or other sends where the data is in RAM in application space and the send is supposed to not block. It first calls lock_socket_buf() to synchronize with other threads a decision about whether a send buffer for the socket already exists in the queue, in which case it must append to that, or whether none exist, in which case it will try to get the socket: trylock_socket(). If it does, it does not have to buffer and can just write straight to TCP, but will then be on the hook to flush any subsequent buffered locks while it had the lock_socket(). If it doesn't get the lock, it will just make a new buffer for the socket and add it to the send queue. Until the current socket holder finishes, all subsequent threads have to add to that one so sequence is respected.

Implementation

varmap.h

This provides a persistent vector like buffer, much like STL::vector<char>, which instead of malloc() uses mmap() and instead of realloc() uses mremap() on a file. It will use ftruncate to grow the file and remap as need be.

i486.h and usersem.h

This implements a semaphore / locking library using the i486(and later) assembly CMPXCHG instruction to handle uncontended locks in user space and makes the necessary futex() calls in the contended case.

Engine

Implemented in pubsub-engine.h and .cpp, the engine is the component that handles the disk, writing a new in RAM message to a stream, and the sending of history and newly arrived push data to subscriber sockets.

It does not ever read a socket. In other words it has client sockets passed to it by the caller who opens, reads, and manages, and closes them, strictly to write stream data to downstream sockets.

It is also unaware of any other servers. The engine itself is not distributed. It is up to the caller to do that.

The interface is in pubsub-engine.h. These are the important functions:

```
int engine_init(struct engine *engine);

int engine_term(struct engine *engine);

void message_create(class vector<char> *stack,
                    const uint32_t stream_id,
                    const void *buf,
                    const size_t len)
int stream_write(struct engine *engine,
                 void *buf,
                 const size_t len,
                 const int src_sock);

//send all history from catchup_from which is which is a

int stream_subscribe(struct engine *engine,
                     const uint32_t stream_id,
                     const int socket,
                     uint64_t catchup_from);
```

```
//stops the stream push to the socket
void stream_unsubscribe(struct engine *engine,
                        const uint32_t stream_id,
                        const int socket);


//unsubscribe socket from all streams
//this is extremely important to call before issuing a close() on
//the socket, otherwise the kernel will reassign the socket for a new
//connection and that connection will start getting pushed data meant
//for the old relic connection that was already closed.
void socket_unsubscribe(struct engine *engine,
                        const int socket);
int noblock_send(struct engine *engine,
                const int socket,
                const void *buf,
                const size_t len);
```

Server

Implemented in pubsub-server.cpp, the server is the component that uses the engine, performing all the threading and networking required to connect multiple engines together with master / slave network topology in the data center and to accept client / admin requests.

The server starts listening with one worker thread on epoll_wait(). As soon as it pops out of epoll, he checks a semaphore to see if there are other worker threads waiting on epoll. If not, it will attempt to make another, unless one is already being made (factory_lock).

Now if what popped out was the listening socket, a new connection is accepted and that goes into epoll also, with the EPOLLONESHOT flag. This means that we don't need to lock anything to read the socket when it pops out of epoll as it will be disabled in epoll and won't pop out for any other threads until we have rearmed it after this request is satisfied.

If a client socket pops out of epoll, we check it for error first, and do a clean shutdown of the socket if need be. Most importantly that means that any reference to the socket in the engine (subscriptions) get destroyed so no more data goes to the socket fd before we close it and it gets reassigned by the kernel.

Since we may not finish reading a complete stream message(downstream) or command(upstream) from the socket, we keep a separate receive buffer for each socket in a hash table like this:

```
unordered_map<int socket, vector<char> *buf> read_queue;
```

If the event is a legitimate read, we empty the TCP buffer into the receive buffer and process as many complete message/commands as we can. We leave any remaining bytes until more bytes come in from TCP.

A note about concurrence on the read_queue: we lock the hash table only to get the pointer to the buffer, then let go of the hash table's lock as the socket itself is essentially locked for read, and so therefore is its read buffer – love EPOLLONESHOT.

Then the server checks to see if the socket is upstream or downstream. As epoll gives us 64 bits of data and the socket fd is only 32, we have another 32 bits to play with without needing our own container. If it is upstream, then we are expecting stream messages on a slaved stream. If it is downstream, then we expect commands being sent to our sever by a client to which we can stream. In the case of commands, they are terminated with:

```
const char cmd_term[] = { 13, 10 };
```

I chose this simply because my telnet client submits this as an end of line when I hit ENTER. So it makes it convenient to use telnet to manage / test the servers.

The following are the list of commands. See below for parameter descriptions.

pub _stream_id_ |Hello World!

Publish everything after the bar "|".

sub _stream_id_ _from_

Catchup from _from_ on all existing history and send new messages with no gap in between and in sequence. Remember that doing this through your telnet session will start sending binary message headers back to you which may not look so hot in your terminal.

close

Kill this connection.

Further, the following administrative commands are available if the connection is local (127.0.0.1). This is a pseudo security, to ensure that at least one has ssh access to the box before they can modify server behaviour:

master _stream_id_

You are responsible for this stream. So if you get publish commands for new messages to it, put them in.

unmaster _stream_id_

The opposite of master. This is the default for every stream when the server starts.

slave _host_ _port_ _stream_id_ _from_

Slave this stream from another master or slave. Forward publish commands upstream and accept subscriptions to this slave (subscribed) stream.

unslave _stream_id_

Cancel slave.

quit

Bring down the server.

Arguments:

_stream_id_

currently an integer between 0 and N_STREAMS-1.

host

like 127.0.0.1

port

like 5050

from

unix timestamp BUT in 64 bit microseconds.

Usage

Build

./make-pubsub-server to compile in debug mode.

Run and Test

Note: never run 2 instances of the server in the same directory. The files will clober each other. It doesn't check for this yet. It should. This would be pretty nasty to debug.

As argument, the server takes the port on which to listen.

Terminal 1	Terminal 2	Terminal 3	Terminal 4
mkdir aa	mkdir bb		
cd aa	cd bb		
../server 5050	../server 5060		
		telnet 127.0.0.1 5050	telnet 127.0.0.1 5060
		master 0	
			Slave 127.0.0.1 5050 0 0
		pub 0 |Hello world	
			pub 0 |Bye world

Both “Hello world” and “Bye world” should appear in both terminals 1 and 2 now, though they took very different paths to get there.

Maintenance

No feature has been added specifically for trimming old obsolete data. But it is pretty easy to do this with a local subscription.

1. Run a new server locally.
2. Slave the master but as of a desired point in time instead of 0.
3. Bring down the master.
4. Unslave the slave. Promote it to master.