

Introduction

This is a repeat of an architecture I worked on several years ago for a high performance/ real-time financial risk management system involving not-unsimilar scaling problems to what Twitter is doing. It works, and it works well.

This post is inspired by the Dare Obasanjo post (<http://www.25hoursaday.com/weblog/2008/05/23/SomeThoughtsOnTwittersAvailabilityProblems.aspx>) about Twitter's scaling issue, and by noting the major saving grace – most calls are looking for the last 20 items, so it doesn't necessarily matter that Scoble has 21,000+ followers because he's never going to see the damned posts also.

This article is just a sketch of how this would work; I acknowledge the devil is in the details. Onaswarm does not use this architecture and will hit similar walls as Twitter if it ever is used at its level. And obviously, it's all hypothetical since I'm not actually doing this. If I was implementing this I would be doing it in Java or C++.

Overview of the problem

- Twitter sucks, probably because of certain pathological edge cases requiring either large numbers of reads or large numbers of writes, or both

Overview of the solution

In case you don't want to read everything below, here's what we end up with:

- one 64Gb machine that stores recent timelines for 30m+ users
- one 64Gb machine that stores last 200 million messages
- a switch Gb Ethernet connection between machines
- a number of smaller machines for easier tasks
- uni-directionally pass messages from machine to machine
- messages can be batched, pipelined, are small not exceeding 4K
- 10,000 requests per second (pending further analysis review)

Overview of how it works

- one-way message passing along always-open connections
- one processor per-major task
 - keep code in the instruction cache at all costs

- the core is a few dedicated machines passing messages across the network
- messages are very small – a few hundred bytes to 4K. Multiple messages can (and will be) sent all at once.

Overview of benefits

- horizontally scalable – just keep adding machines for load, replicating the entire setup if need be
- this should work to about 30 million users
- after about 30 million users another level is needed, but this doesn't overly complicate
- incredibly high throughput

Overview of assumptions

- every message has a unique ID (MID) that fits in 8 bytes
- every user has a unique ID (UID) that fits in 4 bytes
- the average number of friends per-user is 64 (<http://www.kottke.org/07/03/twitter#26563>)
- twitter messages are 140 bytes
- there's a couple of big-assed machines with lots and lots of memory (64Gb) and fast efficient (Gb) network connections; these machines can be purchased for about \$20,000.
- favor frequent queries, service infrequent queries, punish highly infrequent or improbably queries
- there may be a throttling mechanism added to this
 - this could be requiring an ACK/NACK every so many messages, or
 - a separate channel between components
- There's a lot more going on in the API which we do not address here for brevity – in particular:
 - friend list modifications
 - notification on keywords
 - account information has to be looked up!

Networking assumptions

- Gb Ethernet
- 4K max messages
- $1024 * 1024 * 1024 / 8 / 4096 = 32768$ messages/second – let's call it 10,000 because we won't probably get peak speed

Servers / Components

Every server (except the FES) has three types of thread:

- reader threads
- one "task" thread that does stuff that the reader asks and places the result on the writer thread's queue (might use one per core)
- writer threads

FES

The "Front End Server" – i.e. probably an HTTP process that accepts web and API requests. There are lots and lots and lots of these, as needed.

CS

The "Concentration Server" – there is one of these per N FESs, where N is something like 1024. The CS:

- accepts requests from the FES
- pumps the requests into the start of the pipeline
- gets results from the end of the pipeline
- passes the results back to the appropriate FES

These can be simple machines with a couple of Gb of memory.

DBWS

The Database Write Server:

- writes new messages to the DB, returning the MID
- passes on messages

The exact ratio for CS:DBWS or DBWS:TLS will have to be discovered by experiment, though I would probably base it on the first ratio.

These can be simple machines with a couple of Gb of memory.

TLS

The "Time Line Server" – this stores:

- the last N entries for *every* user, where N is something like 128
- the friends list for *every* user

On a 64Gb machine:

- 100 (average number of friends) * 4 (size of UID) = 400 bytes is needed per account to store friends
- 128 (size of TL we are storing) * 12 (size of UID + MID) = 1536 bytes is needed per user

So let's say we need 2K per user – that's 33 million users. Scaling beyond this will require another level of merge sorting, but 33 million is a good start.

MCS

The Message Cache Server – this stores the last M messages via a dictionary/hash table, where M is a big number.

On a 64Gb machine, assuming 256 bytes is needed per-message we are looking at storing 268 million messages for fast retrieval!

DBRS

The DB Read Server – this backfills information the MCS could not retrieve.

The exact ratio for CS:DBWS or DBWS:TLS will have to be discovered by experiment, though I would probably base it on the first ratio.

These can be simple machines with a couple of Gb of memory.

Operations

Add Twit

There's probably no need for the complete round trip I've outlined in this flow below

- FES
 - accepts "add twit" message
 - gets the UID for the user
 - sends message to the CS
 - waits for a result on it's two way socket (this is the only two way connection)
- CS
 - accepts "add twit" message
 - sends message to the DBWS
- DBWS
 - accepts "add twit" message
 - writes the DB

- adds the MID to the message
 - calls the TLS
- TLS
 - accepts "add tweet" message
 - updates user's timeline with the MID and update time, tossing off the oldest entry in the TL
 - NOTE that the text of the tweet is not stored!
 - calls the MCS
- MCS
 - accepts the "add tweet" message
 - adds the tweet to the cache
 - randomly or cleverly removes an old tweet from the cache if memory is full
 - calls the CS
- CS
 - accepts the "add tweet" message; it knows that this is a completed operation
 - sends the result (i.e. "OK") back to the FES

And we're finished.

Computations

Note that computations involved:

- CS: hash table operations, in memory
- DBWS: 1 DB operation, memory + disk
- TLS: $O(1)$ index table operations, in memory
- MCS: $\sim O(1)$ hash table operations, in memory

And also note the size of the message is almost certainly under 256 bytes. Note that the amount of work is constant, no matter if you are JoeBlow-like or Scoble-like.

Get Timeline

- FES
 - accepts "get timeline" message
 - gets the UID for the user
 - sends message to the CS
 - waits for a result on its two way socket (this is the only two way connection)
- CS
 - accepts "get timeline" message
 - sends message to the TLS
- TLS

- accepts "get timeline" message
- looks up all followers
- start with the first follower, then loop through the remainder:
 - merge sort results
- calls the MCS *with* the 20 MIDs to look up
- MCS
 - accepts the "get timeline" message
 - looks up each MID
 - if all MIDs are found, calls the CS
 - if any MIDs are missing, calls the DBRS
- DBRS (optional)
 - accepts the "get timeline" message
 - looks for all MIDs where the message could not be found
 - does a DB search for them and add them
 - calls the CS
- CS
 - accepts the "get timeline" message; it knows that this is a completed operation
 - sends the result (i.e. all the necessary twitter messages) back to the FES

The DBRS could optionally also route the message through the MCS *again* to make the DBRS write them into cache!

Computations

We'll assume everything's in cache, but if they're not we're adding a single database read that will return 1 to 20 entries.

Normal User

Has:

- 64 followers
- assume average 10 integer date comparisons per timeline to do merge sort

So:

- CS: hash table operations, in memory
- TLS
 - $O(1)$ index table operations, in memory – to find followers
 - 640 integer comparisons
- MCS: 20 $\sim O(1)$ hash table operations, in memory

The message from the TLS to the MCS is less than 256 bytes (4 bytes for the MID, 4 bytes for the MID) * 20.

The message from the MCS to the CS is about 3K (140 byte message * 20 plus overhead)

Scoble-like Reader

Has:

- 10000 followers
- assume average 4 integer date comparisons per timeline

So:

- CS: hash table operations, in memory
- TLS
 - O(1) index table operations, in memory – to find followers
 - 40000 integer comparisons
- MCS: 20 ~O(1) hash table operations, in memory

I.e. the only change from the previous step is the 40,000 integer comparisons, as opposed to 640. These are operating in *process cache* due to the one-machine one-task architecture and will be compute quickly – a multi-GHz machine can probably do *hundreds of thousands* of these per second.

Bottlenecks

Let's assume that we average 5000 operations in the TLS to merge timelines. With the network bottleneck at 10,000 requests per second, we need a CPU that can handle 50 million integer operations / second which seems to be an easy fraction of modern CPUs.